Programming Dense Matrix Libraries: The FLAME Collection from ACM TOMS

Programming Dense Matrix Libraries: The FLAME Collection from ACM TOMS

edited by

Robert A. van de Geijn

The University of Texas at Austin

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data not yet available
Draft, November 2008
This "Draft Edition" allows this material to be used while we sort out through what mechanism we will publish the book.

List of Contributors

A large number of people have contributed, and continue to contribute, to the FLAME project. For a complete list, please visit

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

We list the people who have contributed directly to the papers in this collection below.

Paolo Bientinesi, *RWTH Aachen University*

Ernie Chan, *The University of Texas at Austin*

Kazushige Goto, *The University of Texas at Austin*

John A. Gunnels, *IBM T.J. Watson Research Center*

Brian Gunter, *Technical University Delft*

Fred G. Gustavson, *IBM T.J. Watson Research Center*

Greg M. Henry, *Intel Corp.*

Thierry Joffrain, *National Instruments*

Tze Meng Low, *The University of Texas at Austin*

Margaret E. Myers, *The University of Texas at Austin*

Enrique S. Quintana-Ortí, *Universidad Jaume I*

Gregorio Quintana-Ortí, *Universidad Jaume I*

Field G. Van Zee, *The University of Texas at Austin*

Contents

Preface

As of the creation of this book, thirteen articles related the Formal Linear Algebra Methods Environment (FLAME) project have been published in the *ACM Transactions on Mathematical Software* (ACM TOMS). While we would like to think that each individual article constitutes a contribution to the field, together they provide a roadmap for what we believe to be a better design for such libraries. Since articles published over the span of almost a decade are not likely to be read as a coherent collection, we have assembled them into a single document.

The FLAME Project

The objective of the FLAME project has been to transform the development of dense linear algebra libraries from an art reserved for experts to a science that can be understood by novice and expert alike. The methodology encompasses a new notation for expressing algorithms, a methodology for systematic derivation of algorithms, Application Program Interfaces (APIs) for representing the algorithms in code, and tools for the mechanical derivation, implementation and analysis of algorithms.

As a demonstration of the potential of the approach, these techniques have been used to create a new library for dense and banded linear algebra operations. However, that library, which we call `libflame`, is *not* the topic of this book.

Target Audience

Our book *The Science of Programming Matrix Computations* targets mostly the novice and we consider it a good introduction for undergraduates and beginning graduate students. By contrast, this book targets more seasoned researchers: undergraduate and graduate students as well as experts in high-performance scientific computing. It is meant to impress upon the reader our view that the FLAME methodology is a very viable unified solution to the programmability problem for the domain of dense matrix computations.

The Articles

In this document, the articles are given in the order in which they were submitted, which is slightly different from the order in which they appeared. As we discuss how they are connected, we group them into three sets: a set that together describes the methodology; a set that describes how high performance is attained by the building blocks of high performance, matrix-matrix operations; and a set that discusses new algorithms that resulted from the application of the insights.

The FLAME Methodology The first set of articles describes the core methodology and the tools that were developed as part of the FLAME project.

- John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment.*ACM TOMS*, 27(4):422-455, 2001.

  In this article, we lay out a vision that has subsequently become the FLAME project. Many insights are already presented in this article. We give the notation for expressing algorithms and show how it facilitates the formal derivation of families of algorithms for matrix operations. The article previews what later became the FLAME/C API for representing such algorithms in code. The performance benefits that result from choosing the best algorithm from a family of variants are also demonstrated.

- Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert van de Geijn. The Science of Deriving Dense Linear Algebra Algorithms. *ACM TOMS*, 31(1):1-26, 2005.

  In this article, the process by which an algorithm is discovered for a given linear algebra operation is shown to be systematic to the point where it can be reduced to filling out what we have come to call "The Worksheet". The insights in this article change algorithm development from a fine art that is practiced only by experts to a science that can now be understood and applied by a relative novice.

- Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert van de Geijn. Representing Linear Algebra Algorithms in Code: The FLAME APIs. *ACM TOMS*, 31(1):27-59, 2005.

  The FLAME methodology can be used to derive algorithms hand-in-hand with their proofs of correctness. But the algorithm must still be represented in code. To reduce the chance of introducing coding errors in this translation, we propose Application Programming Interfaces (APIs) that allow the code to closely resemble the algorithms. The resulting APIs allow FLAME to be used in a pedagogical setting as well as to implement production-quality libraries such as the `libflame` library. The API is key to FLAME's practical utility because it preserves and translates correctness from algorithm to implementation.

- Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. van de Geijn, Field G. Van Zee, and Ernie Chan. Programming Matrix Algorithms-by-Blocks for Thread-Level Parallelism. *ACM TOMS*, 36(3), Article 14.

  Large Fix "to appear"

  > One of the goals of the FLAME project has been to make libraries more flexible. This is particularly important given that at this moment in time we do not know what architectures will dominate scientific computing ten or even five years into the future. In this article, we illustrate how the FLAME methodology supports high-level abstraction and separation of concerns, yielding an attractive and powerful solution to the programmability problem for multithreaded architectures like Symmetric MultiProcessors (SMPs) and/or multicore systems.
  >
  > We include this article in the set of core articles because it describes an extension of the FLAME API that supports matrices (hierarchically) stored by blocks, algorithms that compute by blocks, and a runtime system that schedules such algorithms to threads, all of which have become core elements in the FLAME methodology.

The Building Blocks of High Performance Library development in the area of scientific computing has to deliver high performance. The two articles in this set describe how algorithms and architectures interact and how data movement can be amortized near-optimally for matrix-matrix operations, the building blocks of high-performance linear algebra libraries.

- Kazushige Goto and Robert A. van de Geijn. Anatomy of High-Performance Matrix Multiplication, *ACM TOMS*, 34(3): Article 12.

  > This article described Kazushigo Goto's techniques for implementing matrix-matrix multiplication that enable his GotoBLAS library for the Basic Linear Algebra Subprograms (BLAS). Among other things, the paper argues that matrices should be primarily blocked for the L2 cache (as opposed to the conventional wisdom that the L1 cache should be targeted). The resulting simple but effective design has served, and continues to serve, many architectures. It disproves the conventional wisdom that the problem of tuning this operation is so complex that blind tuning must be employed, as in the ATLAS library.

- Kazushige Goto and Robert van de Geijn. High-Performance Implementation of the Level-3 BLAS. *ACM TOMS*, 35(1): Article 3.

  > This article extends Goto's techniques for implementing matrix multiplication to other commonly encountered matrix-matrix operations.

Evidence and New Algorithms In order for a new approach like the FLAME methodology to take hold, solid evidence of its potential must be demonstrated. This final set of articles serves that purpose. In addition to making individual contributions, together they establish the applicable scope of the notation, techniques, and tools.

- Enrique S. Quintana-Ortí and Robert van de Geijn, Formal Derivation of Algorithms: The Triangular Sylvester Equation. *ACM TOMS*, 29(2):218–243, 2003.

  This article appeared in print before "The Science of Deriving Dense Linear Algebra Algorithms" but was rewritten after acceptence so that it is better read after that paper. The systematic derivation methodology is applied to a challenging linear algebra operation: the solution of a triangular Sylvester equation. The result is a family of more than a dozen new high-performance algorithms for this operation. It provides early evidence of the value of systematic derivation.

- Brian Gunter and Robert van de Geijn. Parallel Out-of-Core Computation and Updating of the QR Factorization. *ACM TOMS*, 31(1):60-78, 2005.

  This article describes a novel high-performance parallel algorithm for computing the QR factorization of a matrix that is sufficiently large that it must be stored on disk. It fits into the story by demonstrating once again how the FLAME notation allows complex algorithms to be described and how the Parallel Linear Algebra Package (PLAPACK) API can be extended to closely match the FLAME API, thus providing a compatible API for programming distributed memory architectures. Beyond this, it describes an algorithm-by-blocks that is later used in our article "Programming Matrix Algorithms-by-Blocks for Thread-Level Parallelism". Such algorithms-by-blocks have since become a standard part of libraries that target multicore architectures since they increase opportunities for parallelism.

- Thierry Joffrain, Tze Meng Low, Enrique S. Quintana-Ortí, Robert van de Geijn, and Field Van Zee. Accumulating Householder Transformations, Revisited. *ACM TOMS*, 32 (2):169-179, 2006.

  This article revisits techniques for accumulating Householder transformations so that advantage can be taken of the performance of matrix-matrix operations. These results were first reported by others in the late 1980s and early 1990s, but were somehow "lost". In the context of this book, the article illustrates yet again how the FLAME notation can be used to describe linear algebra algorithms. Interestingly, we discovered the results independent from the earlier articles by recognizing that the conventional algorithm for accumulating such transformations is equivalent to a merging of the algorithm described in this article with the inversion of a triangular matrix. This insight came to us by applying the FLAME methodology to each of these operations separately. (This insight is not mentioned in the article itself.)

- Gregorio Quintana-Ortí and Robert van de Geijn. Improving the Performance of Reduction to Hessenberg Form. *ACM TOMS*, 32(2):180-194, 2006.

  The FLAME notation is used to describe a new algorithm that computes a reduction to Hessenberg form, achieving higher performance by casting more computation in terms of matrix-matrix multiplication. The

FLAME APIs were used to implement the algorithms, allowing very rapid development.

- Field G. Van Zee, Paolo Bientinesi, Tze Meng Low, and Robert A. van de Geijn. Scalable Parallelization of FLAME Code via the Workqueuing Model. *ACM TOMS*, 34(2), 29 pages, 2008.

  Part of the problem with the FLAME API is that it discourages the use of indices. Directives that are part of OpenMP, an API for shared memory multiprocessor programming, generally require a loop index to indicate how a loop can be parallelized. The workqueuing model instead allows tasks to be specified and enqueued, which fits the FLAME/C API nicely. This work is now superceded by the developments described in our paper "Programming Matrix Algorithms-by-Blocks for Thread-Level Parallelism." In that paper it is shown how, in addition, dependencies between tasks can be handled via the FLAME/C API.

- Paolo Bientinesi, Brian Gunter, and Robert van de Geijn. Families of Algorithms Related to the Inversion of a Symmetric Positive Definite Matrix. *ACM TOMS*, 35(1), Article 3.

  In this article, the performance benefits of using the FLAME methodology to derive families of algorithms for the same operation is clearly demonstrated. It is shown that algorithms perform very differently on different architectures, ranging from sequential to SMP parallel to distributed memory parallel. More importantly, it is shown that often different algorithms are superior depending on parameters like architecture, problem size, and implementation details. This suggests that conventional libraries, which often incorporate only one or two algorithms, are at an inherent disadvantage.

- Enrique S. Quintana-Ortí and Robert van de Geijn. Updating an LU Factorization with Pivoting. *ACM TOMS*, 35(2), Article 11.

  This article describes how LU factorization with pivoting can be modified to allow the "updating" of a previously-factored submatrix. The result is very similar to the technique developed for updating a QR factorization in the included article "Parallel Out-of-Core Computation and Updating of the QR Factorization." It provides the key insight for the algorithm-by-blocks that is the prime example in the included article "Programming Matrix Algorithms-by-Blocks for Thread-Level Parallelism."

Other Related Journal Articles

Two articles should also be considered part this collection, but are not for the simple reason that they were not published through ACM Transactions of Mathematical Software.

- Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. *SIAM Journal on Scientific Computing*, 22(5):1762–1771, 2001.

  This is the first journal paper to use what later became the FLAME notation. It also uses the merging of the three sweeps that were traditionally use to compute the inverse of a general matrix into a single sweep as a way to argue that the stability of the new algorithm is similar to that of the traditional algorithm.

- Paolo Bientinesi and Robert A. van de Geijn. The Science of Deriving Stability Analyses. Submitted to the *SIAM Journal on Matrix Analysis and Applications*.

  This paper shows how the systematic derivation of algorithms can be extended to also systematically derive numerical stability analyses for the resulting algorithms. This is significant because the FLAME methodology often yields new algorithms for which numerical stability results must be established.

Further Reading

The individual articles reference many related papers, including our own. Among these, the following documents are featured prominently.

- Robert A. van de Geijn. *Using PLAPACK*. The MIT Press, 2007.

  Many of the insights that led to the FLAME methodology came from our experience building parallel linear algebra libraries for distributed memory architectures. This book discribes the Parallel Linear Algebra Package (PLAPACK) API, which inspired the FLAME notation and API.

- John A. Gunnels. A Systematic Approach to the Design and Analysis of Linear Algebra Algorithms. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-01-44 (Ph.D. Dissertation). December 2001.

  The research described in this dissertation was the first to explore what became the FLAME methodology. It also pioneered the use of Mathematica as a tool for mechanical translation of FLAME-like algorithms to code (targeting distributed memory architectures) and performance estimates.

- Paolo Bientinesi. Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-06-46 (Ph.D. Dissertation). September 2006.

  In this dissertation, it is shown that the methodology described in "The Science of Deriving Dense Linear Algebra Algoritms" is sufficiently systematic that it can be mechanically executed by a tool that can perform algebraic manipulation, like Mathematica. In addition, it is shown that

the methodology can be extended to the derivation of numerical stability results.

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

  This book presents the FLAME methodology for deriving and implementing dense linear algebra operations at a much slower pace, making it appropriate for undergraduates and novices.

- Field G. Van Zee. `libflame` *The Complete Reference.* www.lulu.com/content/5915632/, 2009.

  This is the official reference manual for the `libflame` library.

Additional Information

For additional information regarding the FLAME project and other publications, please visit

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

Acknowledgments

In October 2000, Ron Boisvert, then editor-in-chief of ACM TOMS, approached me and suggested that we start publishing our work in TOMS. That turned out to be an excellent suggestion. We thank Ron for his support over the past decade.

We thank Ardavan Pedram for contributing the picture on the cover of this book.

The FLAME project has been generously funded by a number of grants from the National Science Foundation, Microsoft, NEC Systems (America), Inc., and National Instruments. An equipment donation from Hewlett-Packard was also invaluable during the initial phases of the project.

Any opinions, fidings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily re ect the views of the National Science Foundation (NSF).

FLAME: Formal Linear Algebra Methods Environment

JOHN A. GUNNELS
The University of Texas at Austin
FRED G. GUSTAVSON
IBM T.J. Watson Research Center
GREG M. HENRY
Intel Corporation
and
ROBERT A. VAN DE GEIJN
The University of Texas at Austin

Since the advent of high-performance distributed-memory parallel computing, the need for intelligible code has become ever greater. The development and maintenance of libraries for these architectures is simply too complex to be amenable to conventional approaches to implementation. Attempts to employ traditional methodology have led, in our opinion, to the production of an abundance of anfractuous code that is difficult to maintain and almost impossible to upgrade.

Having struggled with these issues for more than a decade, we have concluded that a solution is to apply a technique from theoretical computer science, formal derivation, to the development of high-performance linear algebra libraries. We think the resulting approach results in aesthetically pleasing, coherent code that greatly facilitates intelligent modularity and high performance while enhancing confidence in its correctness. Since the technique is language-independent, it lends itself equally well to a wide spectrum of programming languages (and paradigms) ranging from C and Fortran to C++ and Java. In this paper, we illustrate our observations by looking at the Formal Linear Algebra Methods Environment (FLAME), a framework that facilitates the derivation and implementation of linear algebra algorithms on sequential architectures. This environment

This work was partially supported by the Remote Exploration and Experimentation Project at Caltech's Jet Propulsion Laboratory, which is part of NASA's High Performance Computing and Communications Program, and is funded by NASA's Office of Space Science.
Authors' addresses: J. A. Gunnels, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598; email: gunnels@us.ibm.com; F. G. Gustavson, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598; email: gustav@watson.ibm.com; G. M. Henry, Intel Corp., Bldg. EY2-05, 5350 NE Elam Young Parkway, Hillsboro, OR 97124-6461; email: greg.henry@intel.com; R. A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: rvdg@cs.utexas.edu.

demonstrates that lessons learned in the distributed-memory world can guide us toward better approaches even in the sequential world.

We present performance experiments on the Intel (R) Pentium (R) III processor that demonstrate that high performance can be attained by coding at a high level of abstraction.

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

General Terms: Algorithms; Design; Performance; Theory

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

1. INTRODUCTION

When considering the unmanageable complexity of computer systems, Dijkstra [2000] recently made the following observations:

(i) When exhaustive testing is impossible—i.e., almost always—our trust can only be based on proof (be it mechanized or not).

(ii) A program for which it is not clear why we should trust it, is of dubious value.

(iii) A program should be structured in such a way that the argument for its correctness is feasible and not unnecessarily laborious.

(iv) Given the proof, deriving a program justified by it, is much easier than, given the program, constructing a proof justifying it.

These comments relate to issues that, in context of linear algebra libraries, are orthogonal to the concerns related to numerical stability.

In this paper we make a number of new contributions to the development of linear algebra libraries with regards to the above observations:

—By borrowing from Dijkstra's own contributions to computing science, we show how to systematically derive families of algorithms for a given matrix operation.

—The derivation leads to a structured statement of the algorithms that mirrors how the algorithms are often explained in a classroom setting.

—The derivation of the algorithms provides a proof of their correctness.

—By implementing the algorithms so that the code mirrors the algorithm that is the end-product of this derivational process, opportunities for the introduction of error are reduced. As a result, the proof of the correctness of the algorithm allows us to assert the correctness of the code.

While the resulting infrastructure, the Formal Linear Algebra Methods Environment (FLAME), has allowed us to quickly and reliably implement components of a high-performance linear algebra library, it can equally well benefit library users who need to customize a given routine, or to extend the functionality of their own library.

1.1 Related Work

Advances in software engineering for scientific applications have often been led by libraries for dense linear algebra operations. The first such package to achieve widespread use and to embody new techniques in software engineering was EISPACK [Smith et al. 1976]. EISPACK was also likely the first such package to pay careful attention to the numerical stability of the underlying algorithms. The mid-1970s witnessed the introduction of the Basic Linear Algebra Subprograms (BLAS) [Lawson et al. 1979]. At that time the BLAS were a set of vector operations that allowed libraries to attain high performance on computers possessing a flat memory while remaining highly portable between platforms and simultaneously enhancing modularity and code readability. The first successful library to exploit these BLAS was LINPACK [Dongarra et al. 1979]. By the late 1980s, it was recognized that in order to overcome the gap between processor and memory performance on modern microprocessors, it was necessary to reformulate matrix algorithms in terms of matrix-matrix multiplication-like operations, the level-3 BLAS [Dongarra et al. 1990]. LAPACK [Anderson et al. 1992], first released in the early 1990s, is a high-performance package for linear algebra operations written in terms of the level-3 BLAS. LAPACK offers a functionality that is a superset of LINPACK and EISPACK while achieving high performance on essentially all sequential and shared-memory architectures in a portable fashion.

A major simplification in the implementation of the level-3 BLAS themselves came from the observation that they can be cast in terms of optimized matrix-matrix multiplication [Agarwal et al. 1994; Gustavson et al. 1998b; Kågström et al. 1998]. Further, the performance of the resulting, more portable, system was comparable to the vendor-supplied BLAS in many cases.

With the advent of distributed-memory parallel architectures, a parallel version of LAPACK, ScaLAPACK [Choi et al. 1992], was developed. A major design goal of the ScaLAPACK project was to preserve and re-use as much code from LAPACK as possible. Thus, all layers in the ScaLAPACK software architecture were designed to resemble similar layers in the LAPACK software architecture. It was this decision that complicated the implementation of ScaLAPACK. The introduction of data distribution (across memories) creates a problem analogous to that of creating and maintaining the data structures required for storing sparse matrices. The mapping from indices to matrix element(s) was no longer a simple one. Combining this complication with the monolithic structure of the software led to code that was laborious to construct and is difficult to maintain. The Parallel Linear Algebra Package (PLAPACK) achieves a functionality similar to that of ScaLAPACK, targeting the same distributed-memory architectures while using a FLAME-like approach to hide details related to indexing into and distribution of matrices [van de Geijn 1997]. Indeed, the primary inspiration for FLAME came from earlier work on PLAPACK.

A number of recent library efforts have explored the notion of utilizing hierarchical data structures for storing matrices [Anderson et al. 2000; Gustavson et al. 1998a; Gustavson 1997; 2001]. The central idea is that, by storing matrices by blocks rather than by row- or column-major ordering, data preparation

(copying) for good cache re-use is virtually eliminated. Combining this with recursive and standard algorithms that exploit these data structures, impressive performance improvements have been demonstrated. Notice that more complex data structures for sequential algorithms introduce a complexity similar to that encountered when data is distributed to the memories of a distributed-memory architecture. Since PLAPACK effectively addressed that problem for those architectures, we have strong evidence that FLAME can be extended to accommodate more complex data structures in the context of hierarchical memories.

1.2 A Case for Systematic Derivation and Simplicity of Code

Some would immediately draw the conclusion that a change to a more modern programming language like C++ is a highly desirable, if not a necessary precursor to writing elegant code. The fact is, that most applications that call linear algebra packages are still written in Fortran and/or C. Interfacing such an application with a library written in C++ presents certain complications. However, during the mid-1990s, the Message-Passing Interface (MPI) introduced the scientific computing community to a programming model, object-based programming, that possesses many of the advantages typically associated with the intelligent use of an object-oriented language [Snir et al. 1996]. Using objects (e.g. communicators in MPI) to encapsulate data structures and hide complexity, a much cleaner approach can be achieved.

Our own work on PLAPACK borrowed from this approach in order to hide details of data distribution and data mapping in the realm of parallel linear algebra libraries. The primary concept also germane to this paper is that PLAPACK raises the level of abstraction at which one programs, so that indexing is essentially removed from the code, allowing the routine to reflect the algorithm as it is naturally derived in a classroom setting. Since our initial work on PLAPACK, we have experimented with similar interfaces in such contexts as (parallel) out-of-core linear algebra packages [Gunter et al. 2001; Reiley and van de Geijn 1999] and a low-level implementation of the sequential Basic Linear Algebra Subprograms (BLAS) [Gunnels et al. 2001; Gunnels and van de Geijn 2001b].

One strong motivation for systematically deriving algorithms and reducing the complexity of translating these algorithms to code comes from the fact that for a given operation, a different algorithm may provide higher performance depending on the architecture and/or the problem dimensions. In Gunnels et al. [2001] we showed that the efficient, transportable implementation of matrix-matrix multiplication on a sequential architecture with a hierarchical memory requires a hierarchy of matrix algorithms whose organization, in a very real sense, mirrors that of the memory system under consideration. Perhaps surprisingly, this is necessary even when the problem size is fixed. In the same paper we described a methodology for composing these routines. In this way, minimal coding effort is required to attain superior performance across a wide spectrum of algorithms, architectures, and problem sizes. Analogously, in Gunnels et al. [1998], it was demonstrated that an efficient implementation of parallel matrix

multiplication requires both multiple algorithms and a method for selecting an appropriate algorithm for the presented case, if one is to handle operands of various sizes and shapes. In Gunter et al. [2001] and Reiley [1999], we arrived at a similar conclusion in the context of out-of-core factorization algorithms and their implementation, using the Parallel Out-of-Core Linear Algebra PACKage (POOCLAPACK). To summarize our experiences: as high-performance architectures incorporate cache, local, shared, and distributed memories, all within one system, multiple algorithms for a single operation become necessary for optimal performance. Traditional approaches make the implementation of libraries that span all possibilities, almost impossible.

FLAME is the latest step in the evolution of these systems. We consider FLAME to be an environment in the sense that it encourages the developer to systematically construct a family of algorithms for a given matrix operation. Ideally, the steps that lead to the algorithms are carefully documented, providing the proof that the algorithms are correct. Only after its correctness can be asserted, should the algorithm be translated to code. Since the code mirrors the end-product of the algorithmic derivation process, its correctness can be asserted as well, and minimal debugging and testing will be necessary. Once the code delivers the correct results, functionality can be extended and/or performance optimizations can be incorporated. We illustrate FLAME in the simplest setting, for sequential algorithms. Minor modifications to PLAPACK and POOCLAPACK allow the porting to distributed-memory architectures and/or out-of-core computations with essentially no change to the code.

1.3 Overview

In Section 2 we review some basic insights from formal derivation. In Section 3 we use the LU factorization without pivoting to illustrate the steps we use to develop a family of algorithms for a given operation. In Section 4 we summarize our systematic steps for deriving linear algebra algorithms. In Section 5 we show how library extensions added to the C programming language, together with careful formatting, allow one to write code that reflects the algorithm. The fact that the techniques can be applied to a more difficult operation like LU factorization with partial pivoting is demonstrated in Section 6. In Section 7 we show that it is not the case that high performance is compromised by raising the level of abstraction at which one codes. Future directions and conclusions are discussed in Sections 8 and 9.

2. THE CORRECTNESS OF LOOPS

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

> We prefer to write a while loop using the syntax
>
> **do** $B \rightarrow S$ **od**
>
> where Boolean expression B is called the *guard* and statement S is called the *repetend*.

> [The l]oop is executed as follows: If B 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 B 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
> —$\{P \wedge B\}S\{P\}$ holds—i.e. execution of S begun in a state in which P and B are *true* terminates with P *true*—and
> —$\{P\}$ **do** $B \rightarrow S$ **od** *true*—i.e. execution of the loop begun in a state in which P is *true* terminates.
> Then $\{P\}$ **do** $B \rightarrow S$ **od** $\{P \wedge \neg B\}$ 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 B is *false*].

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

Let us translate the above programming construct into our setting, which we use to accommodate linear algebra algorithms. Consider the loop

while B **do**
  S
enddo

where B is some condition and S is the body of the loop, the above theorem says that

—The loop is entered in a state where some condition P holds, and
—for each iteration, P holds at the top of the loop, and
—the body of the loop S has the property that, if it is executed starting in a state where P holds, it completes in a state where P holds.

Then if the loop completes, it will do so in a state where conditions P and $\neg B$ both hold. Naturally, P and B are chosen such that $P \wedge \neg B$ implies that the desired linear algebra operation has been computed.

A method that formally derives a loop (i.e., iterative implementation) approaches the problem of determining the body of the loop as follows: First, one must determine conditions B and P. Next, the body S should be developed so that it maintains condition P while making progress towards completing the iterative process (eventually B should become *false*). The operations that comprise S follow naturally from simple manipulation of equalities and equivalences using matrix algebra. Thanks to the fundamental invariance theorem, this approach implies correctness of the loop.

What we will argue in this paper is that for a large class of dense linear algebra algorithms there is a systematic way of determining different conditions,

P, that allow us develop loops to compute a given linear algebra operation. The different conditions yield different algorithmic variants for computing the operation. We demonstrate this through the example of LU factorization without pivoting. Once we have demonstrated the techniques in this simpler setting, we will also argue in Section 6, although somewhat more informally, the correctness of a hybrid iterative/recursive LU factorization with partial pivoting.

3. A CASE STUDY: LU FACTORIZATION

We illustrate our approach by considering LU factorization without pivoting. Given a non-singular, $n \times n$ matrix, A, we wish to compute an $n \times n$ lower triangular matrix L with unit main diagonal and an $n \times n$ upper triangular matrix U so that $A = LU$. The original matrix A is overwritten by L and U in the process. We will denote this operation by

$$A \leftarrow \hat{A} = \mathrm{LU}(A)$$

to indicate that A is overwritten by the LU factors of A. Because FLAME produces many variants of LU factorization, it is worthwhile to emphasize the fact that, if exact arithmetic is performed, all variants will result in identical results. To see this, assume that $L_1U_1 = L_2U_2$ are two *different* factorizations. Multiplying both sides by L_2^{-1} on the left and U_1^{-1} on the right yields $L = L_2^{-1}L_1 = U_2U_1^{-1} = U$, where L is unit lower-triangular and U upper-triangular. Now, $L = U$ implies $L = U = I$. It follows that $L_1 = L_2$ and $U_1 = U_2$, so our assumption has been contradicted and the proof of uniqueness is complete.

3.1 A Classical Derivation

The usual derivation of an algorithm for the LU factorization proceeds as follows:

Partition

$$A = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right), \quad L = \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array}\right), \quad \text{and} \quad U = \left(\begin{array}{c|c} \upsilon_{11} & u_{12}^T \\ \hline 0 & U_{22} \end{array}\right)$$

Now $A = LU$ translates to

$$\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} \upsilon_{11} & u_{12}^T \\ \hline 0 & U_{22} \end{array}\right) = \left(\begin{array}{c|c} \upsilon_{11} & u_{12}^T \\ \hline l_{21}\upsilon_{11} & l_{21}u_{12}^T + L_{22}U_{22} \end{array}\right)$$

so the following equalities hold:

$$\left(\begin{array}{c|c} \alpha_{11} = \upsilon_{11} & a_{12}^T = u_{12}^T \\ \hline a_{21} = \upsilon_{11}l_{21} & A_{22} = l_{21}u_{12}^T + L_{22}U_{22} \end{array}\right)$$

Thus, we arrive at the following algorithm

—Overwrite α_{11} and a_{12}^T with υ_{11} and u_{12}^T, respectively (no-op).
—Update $a_{21} \leftarrow l_{21} = a_{21}/\upsilon_{11}$.
—Update $A_{22} \leftarrow A_{22} - l_{21}u_{12}^T$.
—Factor $A_{22} \rightarrow L_{22}U_{22}$ (recursively or iteratively).

$$\textbf{partition } A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \textbf{ where } \quad A_{TL} \textbf{ is } 0 \times 0$$

$$\textbf{do until } A_{BR} \textbf{ is } 0 \times 0$$

$$\textbf{repartition } \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & a_{01} & A_{02} \\ \hline\hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) \textbf{ where } \alpha_{11} \textbf{ is a scalar}$$

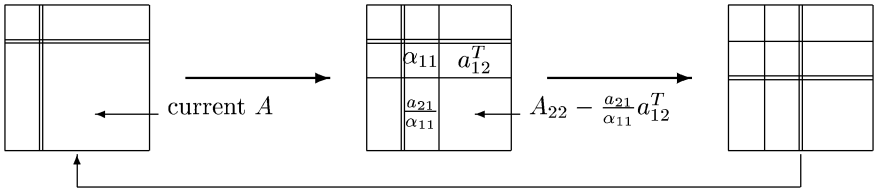
$$\alpha_{11} \leftarrow v_{11} = \alpha_{11} \text{ (no-op)}$$
$$a_{12}^T \leftarrow u_{12}^T = a_{12}^T \text{ (no-op)}$$
$$a_{21} \leftarrow l_{21} = a_{21}/v_{11}$$
$$A_{22} \leftarrow A_{22} - l_{21}u_{12}^T$$

$$\textbf{continue with } \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline\hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

$$\textbf{enddo}$$

Fig. 1. Unblocked lazy algorithm for LU factorization.

The algorithm is usually implemented as a loop, as illustrated in Figure 1. When presented in a classroom setting, this algorithm is typically accompanied by the following progression of pictures:



Here the double lines indicate how far the computation has progressed through the matrix. At the current stage the active part of the matrix resides in the lower-right quadrant of the left picture. Next, the different parts to be updated are identified and the updates given (middle picture). Finally, the boundary that indicates how far the computation has progressed is moved forward (right picture). It is this sequence of three pictures that we will try to capture in the derivation, the specification, and implementation of the algorithm.

3.2 But What is the Loop-Invariant?

Notice that in the above algorithm the original matrix is overwritten by intermediate results until it finally contains L and U. Let $\hat{A}$ indicate the matrix in which the LU factorization is computed, keeping in mind that $\hat{A}$ overwrites A as part of the algorithm. Notice that after k iterations of the algorithm in Figure 1, $\hat{A}$ contains a partial result. We will denote this partial result by $\hat{A}_k$.

In order to prove correctness, one question we must ask is—What intermediate value, $\hat{A}_k$, is in $\hat{A}$ at any particular stage of the algorithm? More precisely, we will ask the question—What are the contents at the beginning of the loop that implements the computation of the factorization (e.g., the loop in Figure 1)?

To answer this question, partition the matrices as follows:

$$A = \left(\begin{array}{c|c} A_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right), \qquad L = \left(\begin{array}{c|c} L_{TL}^{(k)} & 0 \\ \hline L_{BL}^{(k)} & L_{BR}^{(k)} \end{array}\right),$$
$$U = \left(\begin{array}{c|c} U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline 0 & U_{BR}^{(k)} \end{array}\right) \quad \text{and} \quad \hat{A}_k = \left(\begin{array}{c|c} \hat{A}_{TL}^{(k)} & \hat{A}_{TR}^{(k)} \\ \hline \hat{A}_{BL}^{(k)} & \hat{A}_{BR}^{(k)} \end{array}\right)$$

where $A_{TL}^{(k)}$, $L_{TL}^{(k)}$, $U_{TL}^{(k)}$, and $\hat{A}_{TL}^{(k)}$ are all $k \times k$ matrices and "T", "B", "L", and "R" stand for Top, Bottom, Left, and Right, respectively.

Notice that

$$\left(\begin{array}{c|c} A_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c|c} L_{TL}^{(k)} & 0 \\ \hline L_{BL}^{(k)} & L_{BR}^{(k)} \end{array}\right) \left(\begin{array}{c|c} U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline 0 & U_{BR}^{(k)} \end{array}\right)$$
$$= \left(\begin{array}{c|c} L_{TL}^{(k)} U_{TL}^{(k)} & L_{TL}^{(k)} U_{TR}^{(k)} \\ \hline L_{BL}^{(k)} U_{TL}^{(k)} & L_{BL}^{(k)} U_{TR}^{(k)} + L_{BR}^{(k)} U_{BR}^{(k)} \end{array}\right)$$

so that the following equalities must hold:

$$A_{TL}^{(k)} = L_{TL}^{(k)} U_{TL}^{(k)} \tag{1}$$
$$A_{TR}^{(k)} = L_{TL}^{(k)} U_{TR}^{(k)} \tag{2}$$
$$A_{BL}^{(k)} = L_{BL}^{(k)} U_{TL}^{(k)} \tag{3}$$
$$A_{BR}^{(k)} = L_{BL}^{(k)} U_{TR}^{(k)} + L_{BR}^{(k)} U_{BR}^{(k)} \tag{4}$$

We now show that different conditions on the contents of $\hat{A}$ dictate different algorithmic variants for computing the LU factorization, and that these different conditions can be systematically generated from equations 1–4.

Notice that in equations 1–4 the following partial results towards the computation of the factorization can be identified:

$$L\backslash U_{TL}^{(k)}, \quad L_{BL}^{(k)}, U_{TR}^{(k)}, \quad L_{BL}^{(k)} U_{TR}^{(k)}, \quad \text{and} \quad L\backslash U_{BR}^{(k)}$$

Here we use the notation $L\backslash U$ to denote lower and upper triangular matrices that are stored in a square matrix by overwriting the lower and upper triangular parts of that matrix. Recall that L has ones on the diagonal, which need not be stored. We restrict our study to algorithms that employ Gaussian elimination and do not involve redundant computations. Further, we require that one or more of the partial results contributing to the final computation have been computed. A few observations:

—If $L_{TL}^{(k)}$ has been computed, the elements of $U_{TL}^{(k)}$ has been computed as well.

—Since $L_{BL}^{(k)} = A_{BL}^{(k)} U_{TL}^{(k)-1}$, data dependency considerations imply that $U_{TL}^{(k)}$ must be computed before $L_{BL}^{(k)}$.

—Similarly, since $U_{TR}^{(k)} = L_{TL}^{(k)-1} A_{TR}^{(k)}$, data dependency analysis implies that $L_{TL}^{(k)}$ needs to be computed before $U_{TR}^{(k)}$.

—Since the computation overwrites A, if $L_{BL}^{(k)} U_{TR}^{(k)}$ has been computed, $\hat{A}_{BR}^{(k)}$ must contain $A_{BR}^{(k)} - L_{BL}^{(k)} U_{TR}^{(k)}$.

Table I. Possible Loop-Invariants for LU Factorization without Pivoting

Condition	$\hat{A}_k$ contains
No computation has occurred.	$\left(\begin{array}{c\|\|c} A_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline\hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right)$
Only (1) is satisfied.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline\hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right)$
Only (1) and (2) have been satisfied.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right)$
Only (1) and (3) have been satisfied.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline\hline L_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right)$
Only (1), (2), and (3) have been satisfied.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline L_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right)$
(1), (2), and (3) have been satisfied and as much of (4) has been computed *without computing any part of* $L_{BR}^{(k)}$ *or* $U_{BR}^{(k)}$.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline L_{BL}^{(k)} & A_{BR}^{(k)} - L_{BL}^{(k)} U_{TR}^{(k)} \end{array}\right)$
(1), (2), (3), and (4) have all been satisfied.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline L_{BL}^{(k)} & L\backslash U_{BR}^{(k)} \end{array}\right)$

—If $L_{BR}^{(k)}$ has been computed, we assume that $U_{BR}^{(k)}$ has been computed as well (see first bullet).

—If $L\backslash U_{BR}^{(k)}$ has been computed, $A_{BR}^{(k)} - L_{BL}^{(k)} U_{TR}^{(k)}$ must have been computed first.

Taking into account the above observations, we give possible contents of $\hat{A}_k$ in Table I. The first and last conditions indicate that no computation has been performed, or that the final result has been computed, neither of which is a reasonable condition to maintain as part of the loop. This leaves five loop-invariants which, we will see, lead to five different variants for LU factorization.

Note that in this paper, we will not concern ourselves with the question of whether the above conditions exhaust all possibilities. However, they do give rise to many commonly discussed algorithms. In fact, in Dongarra et al. [1984] six variants, called the ijk orders, of $A = LU$ are listed. The jki form is commonly known as a left-looking algorithm while the ikj form is left-looking on A^T. Together, they correspond to the row- and column-lazy variants discussed in this paper. The kij and kji forms both correspond to what has been traditionally called the right-looking algorithm; here, both would be deemed forms of the eager algorithm, one a column, and one a row-oriented version. The ijk and jik forms are more commonly known as the Doolittle (Crout) algorithm and correspond to row and column-oriented versions of the row-column-lazy variant considered in this document. The lazy algorithm discussed in this paper has no corresponding variant in the ijk family of algorithms. Further, the conditions delineated above yield all algorithms depicted on the cover of, and discussed in, G.W. Stewart's [1998] recent book on matrix factorization. This comes as no surprise since we, like Stewart, have adopted some common implicit assumptions about both matrix partitioning and the nature of algorithmic advancement. Our

a priori assumptions were only slightly less constricting than those imposed by the authors who employed the ijk scheme mentioned above. In this paper we have restricted ourselves to a consideration of only those algorithms whose progress is "simple". That is, each iteration of the algorithm is geographically monotonic and formulaically identical. The combination of these two properties leads to algorithms whose (inductive) proofs of correctness are straightforward and whose implementations, given our framework, are virtually foolproof.

We will label any algorithm "Lazy" if it does the least amount of computation possible in the inductive step and "Eager" if it performs as much work as possible at that point. We explain our classification further in Gunnels and van de Geijn [2001a]. It needs to be evaluated against a large class of algorithms before we make any definitive claims regarding is usefulness.

3.3 Lazy Algorithm

This algorithm is often referred to in the literature as a bordered algorithm. Stewart [1998], rather colorfully, refers to it as Sherman's march.

Unblocked Algorithm. Let us assume that only (1) has been satisfied. To determine the body of the loop (statement S), the question becomes how to update the contents of $\hat{A}$:

$$\left(\begin{array}{c|c} \hat{A}_{BR}^{(k)} & \hat{A}_{TR}^{(k)} \\ \hline \hat{A}_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{BR}^{(k)} & A_{TR}^{(k)} \\ \hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right)$$
$$\rightarrow \left(\begin{array}{c|c} \hat{A}_{BR}^{(k+1)} & \hat{A}_{TR}^{(k+1)} \\ \hline \hat{A}_{BL}^{(k+1)} & \hat{A}_{BR}^{(k+1)} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{BR}^{(k+1)} & A_{TR}^{(k+1)} \\ \hline A_{BL}^{(k+1)} & A_{BR}^{(k+1)} \end{array}\right)$$

To answer this, repartition

$$\left(\begin{array}{c|c} A_{TL}^{(k)} & \hat{A}_{TR}^{(k)} \\ \hline A_{BL}^{(k)} & \hat{A}_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c|c} A_{00}^{(k)} & \left(\begin{array}{c|c} a_{01}^{(k)} & A_{02}^{(k)} \end{array}\right) \\ \hline \left(\begin{array}{c} a_{10}^{(k)T} \\ \hline A_{20}^{(k)} \end{array}\right) & \left(\begin{array}{c|c} \alpha_{11}^{(k)} & a_{12}^{(k)T} \\ \hline a_{21}^{(k)} & A_{22}^{(k)} \end{array}\right) \end{array}\right)$$

where $A_{00}^{(k)}$ is $k \times k$ (and thus equal to $A_{TL}^{(k)}$), and $\alpha_{11}^{(k)}$ is a scalar. Repartition $\hat{A}_k$, L, and U similarly. This repartitioning identifies submatrices that must be updated in order to be able to move the boundary (indicated by the double lines) forward. Notice that using this new partitioning, $\hat{A}_k$ currently contains

$$\left(\begin{array}{c|c} L\backslash U_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{00}^{(k)} & \left(\begin{array}{c|c} a_{01}^{(k)} & A_{02}^{(k)} \end{array}\right) \\ \hline \left(\begin{array}{c} a_{10}^{(k)T} \\ \hline A_{20}^{(k)} \end{array}\right) & \left(\begin{array}{c|c} \alpha_{11}^{(k)} & a_{12}^{(k)T} \\ \hline a_{21}^{(k)} & A_{22}^{(k)} \end{array}\right) \end{array}\right)$$

After moving the double lines, the partitioning of A becomes

$$\left(\begin{array}{c|c} A_{TL}^{(k+1)} & A_{TR}^{(k+1)} \\ \hline A_{BL}^{(k+1)} & A_{BR}^{(k+1)} \end{array}\right) = \left(\begin{array}{c|c} \left(\begin{array}{c|c} A_{00}^{(k)} & a_{01}^{(k)} \\ \hline a_{10}^{(k)T} & \alpha_{11} \end{array}\right) & \left(\begin{array}{c} A_{02}^{(k)} \\ \hline a_{12}^{(k)T} \end{array}\right) \\ \hline \left(\begin{array}{c|c} A_{20}^{(k)} & a_{21}^{(k)} \end{array}\right) & A_{22}^{(k)} \end{array}\right)$$

and the partitionings of $\hat{A}_{k+1}$, L, and U change similarly. Thus, $\hat{A}_{k+1}$ must contain

$$\left(\begin{array}{c||c} L\backslash U_{TL}^{(k+1)} & A_{TR}^{(k+1)} \\ \hline\hline A_{BL}^{(k+1)} & A_{BR}^{(k+1)} \end{array}\right) = \left(\begin{array}{c||c} \left(\begin{array}{c|c} L\backslash U_{00}^{(k)} & u_{01}^{(k)} \\ \hline l_{10}^{(k)T} & \upsilon_{11}^{(k)} \end{array}\right) & \left(\begin{array}{c} A_{02}^{(k)} \\ \hline a_{12}^{(k)T} \end{array}\right) \\ \hline\hline \left(\begin{array}{c|c} A_{20}^{(k)} & a_{21}^{(k)} \end{array}\right) & A_{22}^{(k)} \end{array}\right)$$

To summarize, in order to maintain the loop-invariant, the contents of $\hat{A}$ must be updated like

$$\left(\begin{array}{c||c} L\backslash U_{00}^{(k)} & \left(\begin{array}{c|c} a_{01}^{(k)} & A_{02}^{(k)} \end{array}\right) \\ \hline\hline \left(\begin{array}{c} a_{10}^{(k)T} \\ \hline A_{20}^{(k)} \end{array}\right) & \left(\begin{array}{c|c} \alpha_{11}^{(k)} & a_{12}^{(k)T} \\ \hline a_{21}^{(k)} & A_{22}^{(k)} \end{array}\right) \end{array}\right) \rightarrow \left(\begin{array}{c||c} \left(\begin{array}{c|c} L\backslash U_{00}^{(k)} & u_{01}^{(k)} \\ \hline l_{10}^{(k)T} & \upsilon_{11}^{(k)} \end{array}\right) & \left(\begin{array}{c} A_{02}^{(k)} \\ \hline a_{12}^{(k)T} \end{array}\right) \\ \hline\hline \left(\begin{array}{c|c} A_{20}^{(k)} & a_{21}^{(k)} \end{array}\right) & A_{22}^{(k)} \end{array}\right)$$

Thus, it suffices to compute $u_{01}^{(k)}$, $l_{10}^{(k)}$, and $\upsilon_{11}^{(k)}$, overwriting the corresponding parts $a_{01}^{(k)}$, $a_{10}^{(k)}$, and $\alpha_{11}^{(k)}$.

To determine how to compute these quantities, consider

$$\left(\begin{array}{c||c|c} A_{00}^{(k)} & a_{01}^{(k)} & A_{02}^{(k)} \\ \hline\hline a_{10}^{(k)T} & \alpha_{11}^{(k)} & a_{12}^{(k)T} \\ \hline A_{20}^{(k)} & a_{21}^{(k)} & A_{22}^{(k)} \end{array}\right) = \left(\begin{array}{c||c|c} L_{00}^{(k)} & 0 & 0 \\ \hline\hline l_{10}^{(k)T} & 1 & 0 \\ \hline L_{20}^{(k)} & l_{21}^{(k)} & L_{22}^{(k)} \end{array}\right) \left(\begin{array}{c||c|c} U_{00}^{(k)} & u_{01}^{(k)} & U_{02}^{(k)} \\ \hline\hline 0 & \upsilon_{11}^{(k)} & u_{12}^{(k)T} \\ \hline 0 & 0 & U_{22}^{(k)} \end{array}\right)$$

$$= \left(\begin{array}{c||c|c} L_{00}^{(k)}U_{00}^{(k)} & L_{00}^{(k)}u_{01}^{(k)} & L_{00}^{(k)}U_{02}^{(k)} \\ \hline\hline l_{10}^{(k)T}U_{00}^{(k)} & l_{10}^{(k)T}u_{01}^{(k)} + \upsilon_{11}^{(k)} & l_{10}^{(k)T}U_{02}^{(k)} + u_{12}^{(k)T} \\ \hline L_{20}^{(k)}U_{00}^{(k)} & L_{20}^{(k)}U_{01}^{(k)} + l_{21}^{(k)}\upsilon_{11}^{(k)} & L_{20}^{(k)}U_{02}^{(k)} + l_{21}^{(k)}u_{12}^{(k)T} + L_{22}^{(k)}U_{22}^{(k)} \end{array}\right)$$

From this equation we find that the following equalities must hold:

$$\begin{array}{l||l|l} A_{00}^{(k)} = L_{00}^{(k)}U_{00}^{(k)} & a_{01}^{(k)} = L_{00}^{(k)}u_{01}^{(k)} & A_{02}^{(k)} = L_{00}^{(k)}U_{02}^{(k)} \\ \hline\hline a_{10}^{(k)T} = l_{10}^{(k)T}U_{00}^{(k)} & \alpha_{11}^{(k)} = l_{10}^{(k)T}u_{01}^{(k)} + \upsilon_{11}^{(k)} & a_{12}^{(k)T} = l_{10}^{(k)T}U_{02}^{(k)} + u_{12}^{(k)T} \\ \hline A_{20}^{(k)} = L_{20}^{(k)}U_{00}^{(k)} & a_{21}^{(k)} = L_{20}^{(k)}U_{01}^{(k)} + l_{21}^{(k)}\upsilon_{11}^{(k)} & A_{22}^{(k)} = L_{20}^{(k)}U_{02}^{(k)} + l_{21}^{(k)}u_{12}^{(k)T} + L_{22}^{(k)}U_{22}^{(k)} \end{array} \tag{5}$$

To compute $u_{01}^{(k)}$ one must solve the triangular system $L_{00}^{(k)}u_{01}^{(k)} = a_{01}^{(k)}$. The result can overwrite $a_{01}^{(k)}$. To compute $l_{10}^{(k)}$ we solve the triangular system $l_{10}^{(k)T}U_{00}^{(k)} = a_{10}^{(k)T}$. The result can overwrite $a_{10}^{(k)T}$. To determine υ_{11} we merely compute $\upsilon_{11}^{(k)} = \alpha_{11}^{(k)} - l_{10}^{(k)T}u_{01}^{(k)}$. The result can overwrite $\alpha_{11}^{(k)}$. This motivates the algorithm in Figure 2 (left) for overwriting a given non-singular, $n \times n$ matrix A with its LU factorization.

To demonstrate that in deriving the algorithm we have constructively proven its correctness, consider the following:

THEOREM 3.1. *The algorithm in Figure* 2 (*left*) *overwrites a given non-singular*, $n \times n$ *matrix*, A, *with its LU factorization.*

PROOF. To prove this theorem, we merely invoke the Fundamental Invariance theorem. Here the guard B is $A_{BR} \neq 0 \times 0$, predicate P is

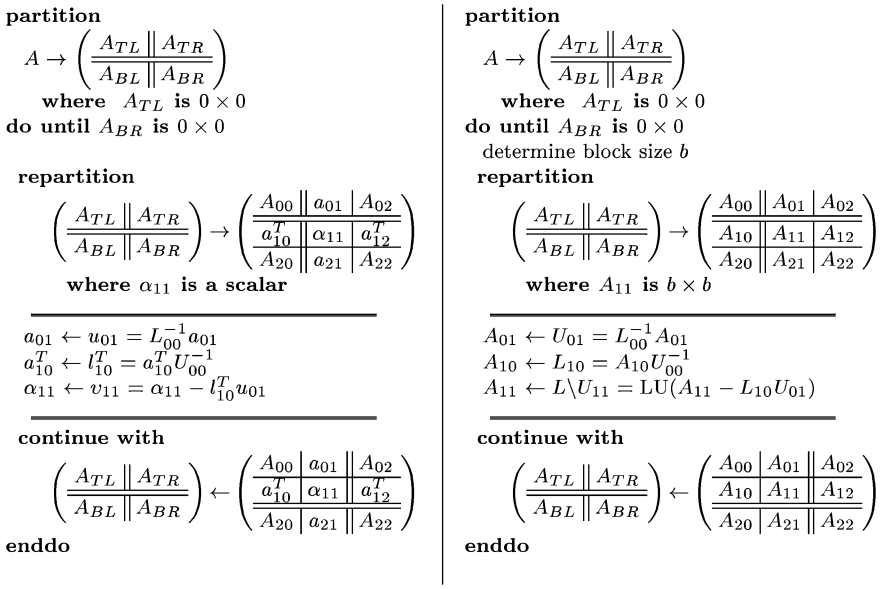


Fig. 2. Unblocked and blocked versions of the lazy variant for computing the LU factorization of a square matrix A (without pivoting).

$$\hat{A} \text{ contains } = \left(\begin{array}{c||c} L\backslash U_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \text{ where } L\backslash U_{TL} \text{ is } k \times k$$

and the statement S is the body of the loop in Figure 2 (left).

First, notice that the statement

$$\text{Partition } A = \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \text{ where } A_{TL} \text{ is } 0 \times 0$$

has the property that after its execution P holds since $L\backslash U_{TL}$, A_{TR}, and A_{BL} are all empty (they have row and/or column dimensions equal to zero) and $A_{BR} = A$. Thus, just before the loop is first entered

$$\hat{A} = \left(\begin{array}{c||c} L\backslash U_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = A_{BR} = A$$

and we conclude that P holds when $k = 0$.

Recall that the body of the loop was developed so that $\{P \wedge B\} S \{P\}$ holds, i.e. if the condition holds at the top of the loop, then it holds at the bottom of the loop (just before the enddo). Also, since at each step the size of A_{BR} decreases by one, guard B will eventually become *false*, $\{P\}$ **do** $B \rightarrow S$ **od** *true* holds (i.e. execution of the loop begun in a state in which P is *true* terminates). We have shown that all of the conditions of the Fundamental Invariance theorem hold. We therefore conclude that if the loop is entered in a state where P holds, it will complete in a state where P is true and guard B is false.

This means that $\hat{A}$ contains $\left(\begin{array}{c||c} L\backslash U_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$ where A_{BR} is 0×0 and completion of the loop transpires when $k = n$. Thus the final contents of the matrix are $\hat{A} = L\backslash U_{TL}$ where L_{TL} and U_{TL} are unit-lower and upper-triangular matrices of order n. We conclude that upon exiting the loop, the matrix has been overwritten by its LU factorization. □

Blocked Algorithm. For performance reasons it, becomes beneficial to derive a blocked version of the above algorithm. The derivation closely follows that of the unblocked algorithm: Again assume that only (1) has been satisfied. The question is now how to compute $\hat{A}_{k+b}$ from $\hat{A}_k$ for some small block size b (i.e. $1 < b \ll n$). To answer this, repartition

$$A = \left(\begin{array}{c||c} A_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline\hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c||c|c} A_{00}^{(k)} & A_{01}^{(k)} & A_{02}^{(k)} \\ \hline\hline A_{10}^{(k)} & A_{11}^{(k)} & A_{12}^{(k)} \\ \hline A_{20}^{(k)} & A_{21}^{(k)} & A_{22}^{(k)} \end{array}\right) \tag{6}$$

where $A_{00}^{(k)}$ is $k \times k$ (and thus equal to $A_{TL}^{(k)}$), and $A_{11}^{(k)}$ is $b \times b$. Repartition L, U, and $\hat{A}_k$ conformally. Notice it is our assumption that $\hat{A}_k$ holds

$$\hat{A}_k = \left(\begin{array}{c||c} L\backslash U_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline\hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c||c|c} L\backslash U_{00}^{(k)} & A_{01}^{(k)} & A_{02}^{(k)} \\ \hline\hline A_{10}^{(k)} & A_{11}^{(k)} & A_{12}^{(k)} \\ \hline A_{20}^{(k)} & A_{21}^{(k)} & A_{22}^{(k)} \end{array}\right)$$

The desired contents of $\hat{A}_{k+b}$ are given by

$$\hat{A}_{k+b} = \left(\begin{array}{c||c} \hat{A}_{TL}^{(k+b)} & \hat{A}_{TR}^{(k+b)} \\ \hline\hline \hat{A}_{BL}^{(k+b)} & \hat{A}_{BR}^{(k+b)} \end{array}\right) = \left(\begin{array}{c|c||c} L\backslash U_{00}^{(k)} & U_{01}^{(k)} & A_{02}^{(k)} \\ \hline\hline L_{10}^{(k)} & L\backslash U_{11}^{(k)} & A_{12}^{(k)} \\ \hline\hline A_{20}^{(k)} & A_{21}^{(k)} & A_{22}^{(k)} \end{array}\right)$$

Thus, it suffices to compute $U_{01}^{(k)}$, $L_{10}^{(k)}$, $L_{11}^{(k)}$, and $U_{11}^{(k)}$.

To derive how to compute these quantities, consider

$$\begin{aligned} A &= \left(\begin{array}{c||c|c} A_{00}^{(k)} & A_{01}^{(k)} & A_{02}^{(k)} \\ \hline\hline A_{10}^{(k)} & A_{11}^{(k)} & A_{12}^{(k)} \\ \hline A_{20}^{(k)} & A_{21}^{(k)} & A_{22}^{(k)} \end{array}\right) = \left(\begin{array}{c||c|c} L_{00}^{(k)} & 0 & 0 \\ \hline\hline L_{10}^{(k)} & L_{11}^{(k)} & 0 \\ \hline L_{20}^{(k)} & L_{21}^{(k)} & L_{22}^{(k)} \end{array}\right) \left(\begin{array}{c||c|c} U_{00}^{(k)} & U_{01}^{(k)} & U_{02}^{(k)} \\ \hline\hline 0 & U_{11}^{(k)} & U_{12}^{(k)} \\ \hline 0 & 0 & U_{22}^{(k)} \end{array}\right) \\ &= \left(\begin{array}{c||c|c} L_{00}^{(k)}U_{00}^{(k)} & L_{00}^{(k)}U_{01}^{(k)} & L_{00}^{(k)}U_{02}^{(k)} \\ \hline\hline L_{10}^{(k)}U_{00}^{(k)} & L_{10}^{(k)}U_{01}^{(k)} + L_{11}^{(k)}U_{11}^{(k)} & L_{10}^{(k)}U_{02}^{(k)} + L_{11}^{(k)}U_{12}^{(k)} \\ \hline L_{20}^{(k)}U_{00}^{(k)} & L_{20}^{(k)}U_{01}^{(k)} + L_{21}^{(k)}U_{11}^{(k)} & L_{20}^{(k)}U_{02}^{(k)} + L_{21}^{(k)}U_{12}^{(k)} + L_{22}^{(k)}U_{22}^{(k)} \end{array}\right) \end{aligned}$$

This yields the equalities

$$\left(\begin{array}{c||c|c} A_{00}^{(k)} = L_{00}^{(k)}U_{00}^{(k)} & A_{01}^{(k)} = L_{00}^{(k)}U_{01}^{(k)} & A_{02}^{(k)} = L_{00}^{(k)}U_{02}^{(k)} \\ \hline\hline A_{10}^{(k)} = L_{10}^{(k)}U_{00}^{(k)} & A_{11}^{(k)} = L_{10}^{(k)}U_{01}^{(k)} + L_{11}^{(k)}U_{11}^{(k)} & A_{12}^{(k)} = L_{10}^{(k)}U_{02}^{(k)} + L_{11}^{(k)}U_{12}^{(k)} \\ \hline A_{20}^{(k)} = L_{20}^{(k)}U_{00}^{(k)} & A_{21}^{(k)} = L_{20}^{(k)}U_{01}^{(k)} + L_{21}^{(k)}U_{11}^{(k)} & A_{22}^{(k)} = L_{20}^{(k)}U_{02}^{(k)} + L_{21}^{(k)}U_{12}^{(k)} + L_{22}^{(k)}U_{22}^{(k)} \end{array}\right) \tag{7}$$

Partition $A = \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0
do until A_{BR} is 0×0

Repartition
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$
where A_{11} is $b \times b$

(a) Eager:
$A_{11} \leftarrow L\backslash U_{11} = \mathrm{LU}(A_{11})$
$A_{12} \leftarrow U_{12} = L_{11}^{-1} A_{12}$
$A_{21} \leftarrow L_{21} = A_{21} U_{11}^{-1}$
$A_{22} \leftarrow A_{22} - L_{21} U_{12}$

(b) Lazy:
View A_{00} as $L\backslash U_{00}$
$A_{01} \leftarrow L_{01} = L_{00}^{-1} A_{01}$
$A_{10} \leftarrow L_{10} = A_{10} U_{00}^{-1}$
$A_{11} \leftarrow L\backslash U_{11} = \mathrm{LU}(A_{11} - L_{10} U_{01})$

(c) Row-lazy:
View A_{00} as $L\backslash U_{00}$
$A_{10} \leftarrow L_{10} = A_{10} U_{00}^{-1}$
$A_{11} \leftarrow L\backslash U_{11} = \mathrm{LU}(A_{11} - L_{10} U_{01})$
$A_{12} \leftarrow U_{12} = L_{11}^{-1}(A_{12} - L_{10} U_{02})$

(d) Column-lazy:
View A_{00} as $L\backslash U_{00}$
$A_{01} \leftarrow U_{01} = L_{00}^{-1} A_{01}$
$A_{11} \leftarrow L\backslash U_{11} = \mathrm{LU}(A_{11} - L_{10} U_{01})$
$A_{21} \leftarrow L_{21} = (A_{21} - L_{20} U_{01}) U_{11}^{-1}$

(e) Row-column-lazy:
$A_{11} \leftarrow L\backslash U_{11} = \mathrm{LU}(A_{11} - L_{10} U_{01})$
$A_{12} \leftarrow U_{12} = L_{11}^{-1}(A_{12} - L_{10} U_{02})$
$A_{21} \leftarrow L_{21} = (A_{21} - L_{20} U_{01}) U_{11}^{-1}$

Continue with
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$$
enddo

Fig. 3. Blocked versions of LU factorization without pivoting for five commonly encountered variants. The different variants share the skeleton that partitions and repartitions the matrix. Executing the operations in one of the five boxes yields a specific algorithm.

Thus,

(1) To compute $U_{01}^{(k)}$ we solve the triangular system $L_{00}^{(k)} U_{01}^{(k)} = A_{01}^{(k)}$. The result can overwrite $A_{01}^{(k)}$.

(2) To compute $L_{10}^{(k)}$ we solve the triangular system $L_{10}^{(k)} U_{00}^{(k)} = A_{10}^{(k)}$. The result can overwrite $A_{10}^{(k)}$.

(3) To compute $L_{11}^{(k)}$ and $U_{11}^{(k)}$ we simply update $A_{11}^{(k)} \leftarrow A_{11}^{(k)} - L_{10}^{(k)} U_{01}^{(k)} = A_{11}^{(k)} - A_{10}^{(k)} A_{01}^{(k)}$ after which the result can be factored into $L_{11}^{(k)}$ and $U_{11}^{(k)}$ using the unblocked algorithm. The result can overwrite $A_{11}^{(k)}$.

The preceding discussion motivates the algorithm in Figure 2 (right) and Figure 3(b) for overwriting the given non-singular, $n \times n$ matrix A with its LU factorization. A careful analysis shows that the blocked algorithm does not incur even a single extra computation relative to the unblocked algorithm.

The proof of the following theorem is similar to that of Theorem 3.1.

THEOREM 3.2. *The algorithm in Figure* 2 (*right*) *overwrites a given non-singular*, $n \times n$ *matrix*, A, *with its LU factorization.*

3.4 Row-Lazy Algorithm

As a point of reference, Stewart [1998] calls this algorithm Pickett's charge south.

Let us assume that only (1) and (2) have been satisfied. We will now discuss only a blocked algorithm that computes $\hat{A}_{k+b}$ from $\hat{A}_k$ while maintaining these conditions.

Repartition A, L, U, and $\hat{A}_k$ conformally as in (6). Our assumption is that $\hat{A}_k$ holds

$$\hat{A}_k = \left(\begin{array}{c||c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c|c||c} L\backslash U_{00}^{(k)} & U_{01}^{(k)} & U_{02}^{(k)} \\ \hline A_{10}^{(k)} & A_{11}^{(k)} & A_{12}^{(k)} \\ \hline\hline A_{20}^{(k)} & A_{21}^{(k)} & A_{22}^{(k)} \end{array}\right)$$

The desired contents of $\hat{A}_{k+b}$ are given by

$$\hat{A}_{k+b} = \left(\begin{array}{c||c} \hat{A}_{TL}^{(k+b)} & \hat{A}_{TR}^{(k+b)} \\ \hline\hline \hat{A}_{BL}^{(k+b)} & \hat{A}_{BR}^{(k+b)} \end{array}\right) = \left(\begin{array}{c|c||c} L\backslash U_{00}^{(k)} & U_{01}^{(k)} & U_{02}^{(k)} \\ \hline L_{10}^{(k)} & L\backslash U_{11}^{(k)} & U_{12}^{(k)} \\ \hline\hline A_{20}^{(k)} & A_{21}^{(k)} & A_{22}^{(k)} \end{array}\right)$$

Thus, it suffices to compute $L_{10}^{(k)}$, $L\backslash U_{11}^{(k)}$, and $U_{12}^{(k)}$. Recalling the equalities in (7) we notice that

(1) To compute $L_{10}^{(k)}$ we can solve the triangular system $L_{10}^{(k)} U_{00}^{(k)} = A_{10}^{(k)}$. The result can overwrite $A_{10}^{(k)}$.

(2) To compute $L_{11}^{(k)}$ and $U_{11}^{(k)}$ we can update $A_{11}^{(k)} \leftarrow A_{11}^{(k)} - L_{10}^{(k)} U_{01}^{(k)} = A_{11}^{(k)} - A_{10}^{(k)} A_{01}^{(k)}$ after which the result can be factored into $L_{11}^{(k)}$ and $U_{11}^{(k)}$. The result can overwrite $A_{11}^{(k)}$.

(3) To compute $U_{12}^{(k)}$ we can update $A_{12}^{(k)} \leftarrow A_{12}^{(k)} - L_{10}^{(k)} U_{02}^{(k)}$ after which we solve the triangular system $L_{11}^{(k)} U_{12}^{(k)} = A_{12}^{(k)}$, overwriting the original $A_{12}^{(k)}$.

These steps, and the preceding discussion, lead one directly to the algorithm in Figure 3(c).

The proof of the following theorem is similar to that of Theorem 3.1.

THEOREM 3.3. *The algorithm in Figure* 3(*c*) *overwrites a given non-singular*, $n \times n$ *matrix*, A, *with its LU factorization.*

3.5 Column-Lazy Algorithm

This algorithm is referred to as a left-looking algorithm in Dongarra et al. [1991] while Stewart [1998] calls it Pickett's charge east.

Let us assume that only (1) and (3) have been satisfied. Now it suffices to compute $U_{01}^{(k)}$, $L\backslash U_{11}^{(k)}$, and $L_{21}^{(k)}$. Using the same techniques as before, one derives the algorithm in Figure 3(d). Again, this algorithm overwrites the given non-singular, $n \times n$ matrix, A, with its LU factorization.

The proof of the following theorem is similar to that of Theorem 3.1.

THEOREM 3.4. *The algorithm in Figure* 3(*d*) *overwrites a given non-singular*, $n \times n$ *matrix*, A, *with its LU factorization.*

3.6 Row-Column-Lazy Algorithm

This algorithm is often referred to as Crout's methods in the literature [Crout 1941].

We assume that only (1), (2), and (3) have been satisfied. This time, it suffices to compute $L\backslash U_{11}^{(k)}$, $U_{12}^{(k)}$, and $L_{21}^{(k)}$, yielding the algorithm in Figure 3(e). Again, this algorithm overwrites a given non-singular, $n \times n$ matrix, A, with its LU factorization.

The proof of the following theorem is similar to that of Theorem 3.1.

THEOREM 3.5. *The algorithm in Figure* 3(*e*) *overwrites a given non-singular*, $n \times n$ *matrix*, A, *with its LU factorization.*

3.7 Eager Algorithm

This algorithm is often referred to as classical Gaussian elimination.

We proceed under the assumption that (1), (2), and (3) have been satisfied, and as much of (4) as possible has been computed, without completing the computation of any part of L_{BR} and U_{BR}. Repartition A, L, U, and $\hat{A}_k$ conformally as in (6). Notice, our assumption is that $\hat{A}_k$ holds

$$\left(\begin{array}{c|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline L_{BL}^{(k)} & A_{BR}^{(k)} - L_{BL}^{(k)} U_{TR}^{(k)} \end{array}\right) = \left(\begin{array}{c|c|c} L\backslash U_{00}^{(k)} & U_{01}^{(k)} & U_{02}^{(k)} \\ \hline L_{10}^{(k)} & A_{11}^{(k)} - L_{10}^{(k)} U_{01}^{(k)} & A_{12} - L_{10}^{(k)} U_{02}^{(k)} \\ \hline L_{20}^{(k)} & A_{21}^{(k)} - L_{20}^{(k)} U_{01}^{(k)} & A_{22}^{(k)} - L_{20}^{(k)} U_{02}^{(k)} \end{array}\right)$$

The desired contents of $\hat{A}_{k+b}$ are given by

$$\left(\begin{array}{c|c} L\backslash U_{TL}^{(k+b)} & U_{TR}^{(k+b)} \\ \hline L_{BL}^{(k+b)} & A_{BR}^{(k+b)} - L_{BL}^{(k+b)} U_{TR}^{(k+b)} \end{array}\right) = \left(\begin{array}{c|c|c} L\backslash U_{00}^{(k)} & U_{01}^{(k)} & U_{02}^{(k)} \\ \hline L_{10}^{(k)} & L\backslash U_{11}^{(k)} & U_{12}^{(k)} \\ \hline L_{20}^{(k)} & L_{21}^{(k)} & A_{22}^{(k)} - L_{20}^{(k)} U_{02}^{(k)} - L_{21}^{(k)} U_{12}^{(k)} \end{array}\right)$$

Thus, it suffices to compute $L\backslash U_{11}^{(k)}$, $L_{21}^{(k)}$, $U_{12}^{(k)}$, and to update $\hat{A}_{22}^{(k)}$. Recalling the equalities in (7) we find

(1) To compute $L_{11}^{(k)}$ and $U_{11}^{(k)}$ we factor $\hat{A}_{11}^{(k)}$ which already contains $A_{11}^{(k)} - L_{10}^{(k)} U_{01}^{(k)}$. The result can overwrite $\hat{A}_{11}^{(k)}$.

(2) To compute $U_{12}^{(k)}$ we update $\hat{A}_{12}^{(k)}$ which already contains $A_{12}^{(k)} - L_{10}^{(k)} U_{02}^{(k)}$ by solving $L_{11}^{(k)} U_{12}^{(k)} = \hat{A}_{12}^{(k)}$, overwriting the original $\hat{A}_{12}^{(k)}$.

(3) To compute $L_{21}^{(k)}$ we update $A_{21}^{(k)}$ which already contains $A_{21}^{(k)} - L_{20}^{(k)} U_{01}^{(k)}$ by solving $L_{21}^{(k)} U_{11}^{(k)} = \hat{A}_{21}^{(k)}$, overwriting the original $\hat{A}_{21}^{(k)}$.

(4) We then update $\hat{A}_{22}^{(k)}$ which already contains $A_{22}^{(k)} - L_{20}^{(k)} U_{02}^{(k)}$ with $\hat{A}_{22}^{(k)} - L_{21}^{(k)} U_{12}^{(k)}$, overwriting the original $\hat{A}_{22}^{(k)}$.

The resulting algorithm is given in Figure 3(a). Notice that this algorithm is the blocked equivalent to the algorithm derived in Section 3.1.

The proof of the following theorem is similar to that of Theorem 3.1.

THEOREM 3.6. *The algorithm in Figure* 3(*a*) *overwrites a given non-singular*, $n \times n$ *matrix*, A, *with its LU factorization.*

4. A RECIPE FOR DERIVING ALGORITHMS

The derivations of the different algorithmic variants of LU factorization, detailed above, were extremely systematic. The following recipe was used:

(1) State the operation to be performed.
(2) Partition the operands. Notice that some justification is needed for the particular way in which they are partitioned. For LU factorization, this has to do with the fact that blocks of zeroes must be isolated in L and U, since they are triangular matrices.
(3) Multiply out all matrix products corresponding to this partitioning.
(4) Equate the submatrix relations that result from the partitioning of Step 3. These define computations that the algorithm must perform in order to maintain correctness.
(5) Pick a loop-invariant from the set of possible loop-invariants that satisfy the equations given in Step 4. Notice that this loop-invariant plays the role of an induction hypothesis.
(6) From that loop-invariant, derive the steps required to maintain the loop-invariant while moving the algorithm forward in the desired direction. This requires the following substeps:
    (a) Repartition so as to expose the boundaries after they are moved.
    (b) Indicate the current contents for the repartitioned matrices.
    (c) Indicate the desired contents for the repartitioned matrices such that the loop-invariant is maintained.
    (d) Determine the computations required to transform (update) the contents indicated in 6b to those indicated in 6c, (Naturally, it must be verified that these computations are possible).
(7) Update the partitioning of the matrices.
(8) Continue until the partitioning yields the null matrix for the "BR" submatrix.
(9) Classify the algorithm. We have developed a systematic way of classifying the derived algorithms based upon the nature of the inductive step. While we used this classification in the labeling of the algorithms in the previous section, we will not go into further detail here.

A more complete recipe for a broader class of linear algebra operations can be found in Gunnels and van de Geijn [2001a].

We again point out that the recipe implicitly provides a proof of correctness for the algorithm, since Steps 5–6d emulate the proof by mathematical

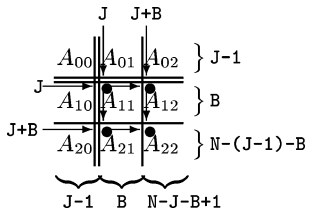


Fig. 4. Partitioning of matrix A with all dimensions annotated when $A_{00} = A_{TL}$ is $(j-1)\times(j-1)$.

induction. Further, the technique employed for deriving these variants of LU factorization generalizes to other factorization algorithms, for example, Cholesky and QR.

5. ENCODING THE ALGORITHM IN C

In this section we briefly discuss how dense linear algebra algorithms, as presented in Figures 1–3, can be translated into code. We first show a more traditional approach, as it appears in popular packages like LAPACK. Next, we present an alternative framework that allows implementation at a higher level of abstraction that mirrors how we naturally present the algorithms. This second approach has been successfully used in PLAPACK; our FLAME framework represents a refinement of this methodology.

5.1 Classic Implementation with the BLAS

Let us consider the blocked eager algorithm for the LU factorization presented in Figure 3(a). This algorithm requires an LU factorization of a small matrix, $A_{11} \leftarrow L\backslash U_{11} = \text{LU fact.}(A_{11})$, triangular solves with multiple right-hand-sides to update $A_{12} \leftarrow U_{12} = L_{11}^{-1}A_{12}$ and $A_{21} \leftarrow L_{21} = A_{21}U_{11}^{-1}$, and a matrix-matrix multiply to update $A_{22} \leftarrow A_{22} - L_{21}U_{12}$. The triangular solves and matrix-matrix multiply are part of the Basic Linear Algebra Subprograms (BLAS) (calls to the routines `DTRSM` and `DGEMM`, respectively). To understand this code, it helps to consider the partitioning of the matrix for a typical loop index j, as illustrated in Figure 4: A_{11} is `B` by `B` and starts at element `A(J,J)`, A_{21} is `N-(J-1)-B` by `B` and starts at element `A(J+B,J)` , A_{12} is `B` by `N-(J-1)-B` and starts at element `A(J,J+B)`, and A_{22} is `N-(J-1)-B` by `N-(J-1)-B` and starts at element `A(J+B,J+B)`. The resultant code is given in Figure 5.

Given this picture, it is relatively easy to determine all of the parameters that must be passed to the appropriate BLAS routines.

5.2 The Algorithm *is* the Code

We would argue that it is relatively easy to generate the code in Figure 5 given the algorithm in Figure 3(a) and the picture in Figure 4. However, the translation of the algorithm to the code is made tedious and error-prone by the fact that one has to think very carefully about indices and matrix dimensions. While this

```
      INTEGER           N, LDA, NB, J, B
      DOUBLE PRECISION  A( LDA, * ), ONE, NEG_ONE
      PARAMETER         ( ONE = 1.0D00, NEG_ONE = -1.0D00 )

      DO J=1, N, NB
         B = MIN( N-J+1, NB )
C                                                 A11 <- L\U11 = LU fact( A11 )
         CALL LU_EAGER_LEVEL2( B, A( J,J ), LDA )

         IF ( J+B <= N ) THEN
C                                                 A12 <- U12 = inv( L11 ) * A12

            CALL DTRSM("LEFT",  "LOWER TRIANGULAR", "NO TRANSPOSE", "UNIT DIAGONAL",
     $                 ONE, B, N-J-B, A( J,J ), LDA, A( J, J+B ), LDA)

C                                                 A21 <- L21 = A21 * inv( U11 )

            CALL DTRSM("RIGHT", "UPPER TRIANGULAR", "TRANSPOSE", "NONUNIT DIAGONAL",
     $                 ONE, N-J-B, B, A( J,J ), LDA, A( J+B, J ), LDA)

C                                                 A22 <- A22 - A21 * A12

            CALL DGEMM("NO TRANSPOSE", "NO TRANSPOSE", N-(J-1)-B, N-(J-1)-B, B,
     $           NEG_ONE, A( J+B, J ), LDA, A( J, J+B ), LDA, ONE, A( J+B, J+B), LDA)
         ENDIF
      ENDDO

      RETURN
      END
```

Fig. 5. Fortran implementation of blocked eager LU factorization algorithm using the BLAS. (Find the bug without referring to Fig. 4 or the text!).

is not much of a problem when implementing just one algorithm, real difficulties may arise when implementing a number of possible algorithmic variants for a given operation, or, in the case of a library such as LAPACK, implementing even a single such variant of each of a large number of operations. One becomes even more acutely aware of these issues when distributed-memory architectures enter the picture, as in ScaLAPACK.

In an effort to make the code look like the algorithms given in Figure 3, while simultaneously accounting for the constraints imposed by C and Fortran, we have developed FLAME. The algorithmic and code skeletons shared by the five variants for the LU factorization, developed earlier in this paper, are given in Figures 6 and 7, respectively. To understand the code, it suffices to realize that A is being passed to the routine as a data structure, `A`, that describes all attributes of this matrix, such as dimensions and method of storage. Inquiry routines like `FLA_Obj_length` are used to extract information, in this case the row dimension of the matrix. Finally, `ATL`, `A00`, etc. are simply references into the original array described by `A`.

If one is familiar with the coding conventions used to name the BLAS kernels, it is clear that the following code segments, when entered in the appropriate place (lines 22–34) in the code in Figure 7, implement the different variants of the LU factorization:

$$
\begin{array}{l}
\text{Partition } A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \\
\quad \text{where } A_{TL} \text{ is } 0 \times 0 \\
\text{do until } A_{BR} \text{ is } 0 \times 0 \\
\quad \text{Repartition} \\
\qquad \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) \\
\qquad \text{where } A_{11} \text{ is } b \times b \\
\hline
\qquad \text{insert update here} \\
\hline
\quad \text{Continue with} \\
\qquad \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) \\
\text{enddo}
\end{array}
$$

Fig. 6. Algorithm skeleton for LU factorization without pivoting.

Lazy algorithm

```
23 FLA_Trsm(FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
24          ONE, A00, A10);
25 FLA_Trsm(FLA_LEFT, FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
26          ONE, A00, A01);
27 FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A01, ONE, A11);
28 FLA_LU_nopivot_level2(A11);
```

Row-lazy algorithm

```
23 FLA_Trsm(FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
24          ONE, A00, A10);
25 FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A01, ONE, A11);
26 FLA_LU_nopivot_level2(A11);
27 FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A02, ONE, A12);
28 FLA_Trsm(FLA_LEFT, FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
29          ONE, A11, A12);
```

Column-lazy algorithm

```
23 FLA_Trsm(FLA_LEFT, FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
24          ONE, A00, A01);
25 FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A01, ONE, A11);
26 FLA_LU_nopivot_level2(A11);
27 FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A20, A01, ONE, A21);
28 FLA_Trsm(FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
29          ONE, A11, A21);
```

Row-column-lazy algorithm

```
23 FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A01, ONE, A11);
24 FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A20, A01, ONE, A21);
25 FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A02, ONE, A12);
26 FLA_LU_nopivot_level2(A11);
27 FLA_Trsm(FLA_LEFT, FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
28          ONE, A11, A12);
```

```
#include "FLAME.h"

void FLA_LU_nopivot_skeleton( FLA_Obj A, nb_alg )
{
  FLA_Obj     ATL, ATR,   A00, A01, A02,
              ABL, ABR,   A10, A11, A12,
                          A20, A21, A22;

  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                  /* ************** */
                    &ABL, /**/ &ABR,
            /* with */ 0, /* by */ 0, /* submatrix */ FLA_TL );

  while ( b=min(min(FLA_Obj_length( ABR ), FLA_Obj_width( ABR )), nb_alg) != 0 )
   {
    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
                         /* ************* */   /* ******************** */
                                /**/            &A10, /**/ &A11, &A12,
                           ABL, /**/ ABR        &A20, /**/ &A21, &A22,
            /* with */ b, /* by */ b, /* A11 split from */ FLA_BR );
    /* ************************************************************************ */

                  insert code for update here

    /* ************************************************************************ */
    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,      A00, A01, /**/ A02,
                                    /**/            A10, A11, /**/ A12,
                             /* ************** */ /* ****************** */
                               &ABL, /**/ &ABR,     A20, A21, /**/ A22,
            /* with A11 added to submatrix */ FLA_TL );
  }
}
```

Fig. 7. A code skeleton for the C implementation of many of the blocked LU factorization algorithms using FLAME.

```
FLA_Trsm(FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
         ONE, A11, A21);
```

Eager algorithm

```
FLA_LU_nopivot_level2( A11 );
FLA_Trsm(FLA_LEFT, FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
         ONE, A11, A12);
FLA_Trsm(FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
         ONE, A11, A21);
FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A21, A12, ONE, A22);
```

5.3 Positive Features of the FLAME Approach

One can argue that determining which of the two methods for coding the algorithms might be deemed “superior”, is simply a matter of taste. However, to support our case, we list the following questions and/or observations:

—What if a bug were introduced into the code in Figure 5? Indeed, in that code we “accidentally” replaced `N - (J - 1) - B` with `N - J - B`. This kind of bug is extremely hard to track down since the only clue is that the code produces the wrong answer or causes a segmentation fault. A similar bug is not as

easily introduced into the code implemented using FLAME since it does not contain indices. Furthermore, with this approach it is easy to perform a run-time check in order to determine if the dimensions of the different matrix operands passed to a routine, are consistent.

—When coding all variants of the LU factorization one inherently has to derive all algorithms, leading to descriptions like those given in Figure 3. However, translating those to code like that given in Figure 5 would require *several* careful considerations of the picture in Figure 4 . Moreover, due to the detailed and extensive indexing involved in that approach, considerable testing would be required before one could declare the code to be bug-free. By contrast, given the algorithms, it has been argued that it is straightforward to generate all variants using FLAME. As has already been mentioned, since the code closely resembles the algorithm, one can be much more confident about its correctness before the code is tested.

—What if we wished to parallelize the given code? Notice that parallelizing a small subset of the functionality of LAPACK as part of the ScaLAPACK project has taken considerable effort. The FLAME code can be transformed into PLAPACK code essentially by replacing `FLA_` by `PLA_`. This highlights the one-to-one correspondence between FLAME and PLAPACK codes; this correspondence is found to be lacking when one considers LAPACK and ScaLAPACK codes in the same light.

—What if we needed a parallel out-of-core version of the code? In principle, the FLAME code can be transformed into Parallel Out-of-Core Linear Algebra PACKage (POOCLAPACK) code by replacing `FLA_` by `POOCLA_`.

5.4 But What About Fortran?

Again using MPI as an inspiration, a Fortran interface is available for FLAME. Examples of Fortran code are available on the FLAME web page, given at the end of this paper.

5.5 Proving the Implementation Correct

In Section 3.3 we proved correctness of the lazy algorithm and in subsequent subsections of Section 3 asserted that the correctness of the other algorithms can be established in much the same way. If the routines called by the described FLAME code correctly implement the operations implied by their names, then it can be argued that the code itself is correct. Indeed, debugging is not necessary.

There are a number of reasons that we feel comfortable in making such a bold assertion. The justifications for the statement rely upon features of both our systematic algorithmic design methodology, the library supporting the implementation of the algorithm, and the relationship between the two.

The manner in which we systematically generate algorithms relies, primarily, on two design pillars, which together make up FLAME. The first is that we have limited the class of problems under consideration to those in linear algebra. The second is that our algorithms consistently build upon the Fundamental Invariance theorem. This restriction leads to the development of algorithms whose correctness can be established.

FLAME is designed to express these systematically generated algorithms in a manner that is both concise and unambiguous. Therefore, the FLAME code can be made to mirror the algorithms thus produced. This leads one to conclude that the two most common sources of error are eliminated. The translation from algorithm to code can be easily automated because of the one-to-one relation between the two, so that a very common mistake can be obviated, namely the code not reflecting the algorithm (when one considers a textual version of the algorithm as it might be presented in a textbook). A second common mistake made with such codes, indexing errors, is eliminated from the top-level expression of FLAME code because FLAME does not do explicit indexing. To be certain, there are a *few* support routines within FLAME that perform indexing. However, these routines are so small, that they are amenable to both standard proof-of-correctness techniques and to truly "exhaustive" testing. In a sense, these routines are analogous to FLAME's "assembly language" and their reliability is comparable to that of a robust compiler.

Because our method of derivation leads to a class of algorithms whose proof of correctness is straightforward and since the language we use to express the produced algorithms should not lead to any (unintentional) mistranslation from algorithm to code, we feel that the *coupled* system leads to programs whose correctness follows from a mathematical derivation of the algorithm.

6. LU FACTORIZATION WITH PARTIAL PIVOTING

We now demonstrate that the techniques that we introduced using the example of LU factorization without pivoting, are also applicable to the case of LU factorization with partial pivoting. The latter algorithm is the one commonly implemented, but involves complications that have traditionally made its derivation coding a more intricate and time-consuming procedure.

6.1 Notation

Let I_m denote the $m \times m$ identity matrix and $\tilde{P}_m(i)$ be the $m \times m$ permutation matrix such that $\tilde{P}_m(i)A$ only swaps the first and ith rows of A. Here, we consider an $m \times n$ matrix, A, where $m \geq n$, and define

$$P_m(p_0, p_1, \ldots, p_{k-1}) = \begin{pmatrix} I_{k-1} & 0 \\ 0 & \tilde{P}_{m-k+1}(p_{k-1}) \end{pmatrix} \cdots \begin{pmatrix} I_1 & 0 \\ 0 & \tilde{P}_{m-1}(p_1) \end{pmatrix} \tilde{P}_m(p_0)$$

and $P_{m;i:j} = P_m(p_i, \ldots, p_j)$. Here p_k equals the index, relative to the top row of the currently active matrix (A_{BR} in previous discussions), of the row that is swapped at the kth step of LU factorization with partial pivoting. Thus $P_m(p_0, p_1, \ldots, p_{k-1})A$ equals the matrix that results after swapping rows 0 and p_0 followed by swapping rows 1 and $p_1 + 1$, and so forth, in that order. Also, $P_{m;i:j}A$ equals the matrix that results after swapping rows i and p_i followed by $i + 1$ and $p_{i+1} + 1$, and so forth, in that order.

It is well-known that LU factorization with partial pivoting produces the LU factorization

$$P_{m;0:n-1}A = LU \tag{8}$$

6.2 Derivation of the Invariants

Now, let us examine the possible contents of matrix $\tilde{A}_k = \mathcal{P}A$, where $\mathcal{P} = P_{m;0:k-1}$, the matrix as it has been overwritten partially into the LU factorization with partial pivoting. Equation 8 is equivalent to

$$\left(\begin{array}{c|c} I_k & 0 \\ \hline 0 & P_{m-k;k:n-1} \end{array}\right) \tilde{A}_k = LU$$

or

$$\tilde{A}_k = \left(\begin{array}{c|c} I_k & 0 \\ \hline 0 & Q^T \end{array}\right) LU$$

where

$$Q = P_{m-k;k:n-1}$$

Partitioning

$$\tilde{A}_k = \left(\begin{array}{c|c} \tilde{A}_{TL}^{(k)} & \tilde{A}_{TR}^{(k)} \\ \hline \tilde{A}_{BL}^{(k)} & \tilde{A}_{BR}^{(k)} \end{array}\right), \quad L = \left(\begin{array}{c|c} L_{TL}^{(k)} & 0 \\ \hline L_{BL}^{(k)} & L_{BR}^{(k)} \end{array}\right), \quad \text{and } U = \left(\begin{array}{c|c} U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline 0 & U_{BR}^{(k)} \end{array}\right),$$

we find that

$$\begin{aligned} \left(\begin{array}{c|c} \tilde{A}_{TL}^{(k)} & \tilde{A}_{TR}^{(k)} \\ \hline \tilde{A}_{BL}^{(k)} & \tilde{A}_{BR}^{(k)} \end{array}\right) &= \left(\begin{array}{c|c} I_k & 0 \\ \hline 0 & Q^T \end{array}\right) \left(\begin{array}{c|c} L_{TL}^{(k)} & 0 \\ \hline L_{BL}^{(k)} & L_{BR}^{(k)} \end{array}\right) \left(\begin{array}{c|c} U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline 0 & U_{BR}^{(k)} \end{array}\right) \\ &= \left(\begin{array}{c|c} L_{TL}^{(k)} U_{TL}^{(k)} & L_{TL}^{(k)} U_{TR}^{(k)} \\ \hline \tilde{L}_{BL}^{(k)} U_{TL}^{(k)} & \tilde{L}_{BL}^{(k)} U_{TR}^{(k)} + \tilde{L}_{BR}^{(k)} U_{BR}^{(k)} \end{array}\right) \end{aligned}$$

where $L_{BL} = Q\tilde{L}_{BL}^{(k)}$ and $L_{BR} = Q\tilde{L}_{BR}^{(k)}$. Thus, for $0 \le k < n$, the equalities in equations 1–4 must again hold, except that $L_{BL}^{(k)}$, $L_{BR}^{(k)}$, and $A^{(k)}$, are now replaced by $\tilde{L}_{BL}^{(k)}$, $\tilde{L}_{BR}^{(k)}$, and $\tilde{A}^{(k)}$, respectively. We mention, as before, that unaccented submatrices of L and U denote final values. As for LU factorization without pivoting, different conditions on the contents of $\hat{A}_k$ logically dictate different variants for computing the LU factorization with partial pivoting. These are given in Table 1, with the provisos mentioned above. Notice that in addition, a necessary condition is that $p_0, \ldots, p_{k-1}$ have been computed.

The second and third conditions listed in Table 1 are impractical since the computation of $p_0, \ldots, p_{k-1}$ requires that the entries of $L_{BL}^{(k)}$ be computed. By taking entries 4 through 6, listed in Table 1, together with the requirement that $p_0, \ldots, p_{k-1}$ have been computed, and using them as part of predicate P, three different variants for LU factorization with partial pivoting can be derived. These conditions again lead to column-lazy (left-looking), row-column-lazy (Crout), and eager (right-looking) variants, respectively, this time with partial pivoting incorporated.

6.3 Derivation of the Eager Algorithm

Let us concentrate on the eager algorithm. Notice, our assumption is that $\hat{A}_k$ holds

$$\hat{A}_k = \left(\begin{array}{c||c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline \tilde{L}_{BL}^{(k)} & \hat{A}_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c||c|c} L\backslash U_{00}^{(k)} & U_{01}^{(k)} & U_{02}^{(k)} \\ \hline\hline \tilde{L}_{10}^{(k)} & \tilde{A}_{11}^{(k)} - \tilde{L}_{10}^{(k)} U_{01}^{(k)} & \tilde{A}_{12}^{(k)} - \tilde{L}_{10}^{(k)} U_{02}^{(k)} \\ \hline \tilde{L}_{20}^{(k)} & \tilde{A}_{21}^{(k)} - \tilde{L}_{20}^{(k)} U_{01}^{(k)} & \tilde{A}_{22}^{(k)} - \tilde{L}_{20}^{(k)} U_{02}^{(k)} \end{array}\right).$$

The desired contents of $\hat{A}_{k+b}$ are given by

$$\begin{aligned} \hat{A}_{k+b} &= \left(\begin{array}{c||c} L\backslash U_{TL}^{(k+b)} & U_{TR}^{(k+b)} \\ \hline\hline \bar{L}_{BL}^{(k+b)} & \hat{A}_{BR}^{(k+b)} \end{array}\right) \\ &= \left(\begin{array}{c|c||c} L\backslash U_{00}^{(k)} & U_{01}^{(k)} & U_{02}^{(k)} \\ \hline L_{10}^{(k)} & L\backslash U_{11}^{(k)} & U_{12}^{(k)} \\ \hline\hline \bar{L}_{20}^{(k)} & \bar{L}_{21}^{(k)} & \bar{A}_{22}^{(k)} - \bar{L}_{20}^{(k)} U_{02}^{(k)} - \bar{L}_{21}^{(k)} U_{12}^{(k)} \end{array}\right) \end{aligned}$$

where, $Q_1 = P_{m-k;k:k+b-1}$, $\bar{A}_{BR}^{(k)} = Q_1 \tilde{A}_{BR}^{(k)}$, and $\left(\begin{array}{c} L_{10}^{(k)} \\ \hline L_{20}^{(k)} \end{array}\right) \leftarrow Q_1 \left(\begin{array}{c} L_{10}^{(k)} \\ \hline L_{20}^{(k)} \end{array}\right)$. Note that $L\backslash U_{11}^{(k)}$ and $L_{21}^{(k)}$ are defined by equation 9, below, and $L_{10}^{(k)} = \bar{L}_{10}^{(k)}$.

With some effort it can be verified that the following updates have the desired effect:

—Compute Q_1, given by $\{p_k, \ldots, p_{k+b-1}\}$, $L_{11}^{(k)}$, $U_{11}^{(k)}$, and $\bar{L}_{21}^{(k)}$ such that

$$\left(\begin{array}{c} \hat{A}_{11}^{(k)} \\ \hline \hat{A}_{21}^{(k)} \end{array}\right) = \left(\begin{array}{c} L_{11}^{(k)} \\ \hline \bar{L}_{21}^{(k)} \end{array}\right) U_{11}^{(k)} \tag{9}$$

overwriting

$$\left(\begin{array}{c} \hat{A}_{11}^{(k)} \\ \hline \hat{A}_{21}^{(k)} \end{array}\right) \leftarrow \left(\begin{array}{c} L\backslash U_{11}^{(k)} \\ \hline \bar{L}_{21}^{(k)} \end{array}\right)$$

—Permute and overwrite: $\left(\begin{array}{c} A_{10}^{(k)} \\ \hline A_{20}^{(k)} \end{array}\right) \leftarrow Q_1 \left(\begin{array}{c} L_{10}^{(k)} \\ \hline L_{20}^{(k)} \end{array}\right)$.

—Permute and overwrite: $\left(\begin{array}{c} A_{12}^{(k)} \\ \hline A_{22}^{(k)} \end{array}\right) \leftarrow Q_1 \left(\begin{array}{c} A_{12}^{(k)} \\ \hline A_{22}^{(k)} \end{array}\right)$.

—Update $\hat{A}_{12}^{(k)} \leftarrow U_{12}^{(k)} = L_{11}^{-1(k)} \hat{A}_{12}^{(k)}$ and $\hat{A}_{22}^{(k)} \leftarrow \hat{A}_{22}^{(k)} - \bar{L}_{21}^{(k)} U_{12}^{(k)}$.

In Figure 8 we show how an eager blocked LU factorization with partial pivoting can be expressed in our algorithmic format. In this algorithm, the operation $\mathrm{LU}_{\mathrm{piv}}(B)$ returns the result of an LU factorization with partial pivoting of matrix B, as well as the pivot indices. In that figure, p_1 is a vector of pivot indices and $P(p_1)$ takes the place of $P_{m-k;k:k+b-1}$.

An unblocked algorithm results when the block size, b, is always chosen to equal unity. In this case, the operation

$$\left[A_{BR}^{(1)}, p_1\right] \leftarrow \left[\left(\begin{array}{c} L\backslash U_{11} \\ \hline L_{21} \end{array}\right), p_1\right] = \mathrm{LU}_{\mathrm{piv}}\left(A_{BR}^{(1)}\right) \tag{10}$$

is replaced by a determination of the index of the element in vector $A_{BR}^{(1)}$, followed by a swap of that element with the first element of that vector, and finally a

$$\text{Partition } A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \text{ and } p = \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$$
where A_{TL} is 0×0 and p_T has 0 elements

do until A_{BR} is 0×0

Determine block size b

Partition
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$
where A_{11} is $b \times b$

Partition
$$\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) = \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$
where p_1 has b elements

Partition
$$A_{BR} = \left(\begin{array}{c|c} A_{BR}^{(1)} & A_{BR}^{(2)} \end{array}\right)$$
where $A_{BR}^{(1)}$ has width b.

$$\left[A_{BR}^{(1)}, p_1\right] \leftarrow \left[\left(\frac{L\backslash U_{11}}{L_{21}}\right), p_1\right] = \mathrm{LU}_{\mathrm{piv}}(A_{BR}^{(1)})$$
$A_{BL} \leftarrow P(p_1) A_{BL}$
$A_{BR}^{(2)} \leftarrow P(p_1) A_{BR}^{(2)}$
$A_{12} \leftarrow U_{12} = L_{11}^{-1} A_{12}$
$A_{22} \leftarrow A_{22} - L_{21} U_{12}$

Continue with
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$
$$\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) = \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

enddo

Fig. 8. Eager blocked LU factorization with partial pivoting.

scaling of the elements of A_{21} by $1/A_{11}$. (Notice that now A_{21} is a vector and A_{11} is a scalar). In other words, the operation in equation 10 is replaced by

$$\begin{array}{l} \text{Choose } p_1 \text{ s.t. } \left|\left[A_{BR}^{(1)}\right]_{p_1}\right| = \max_i \left|\left[A_{BR}^{(1)}\right]_i\right| \\ \text{Swap } \left[A_{BR}^{(1)}\right]_1 \leftrightarrow \left[A_{BR}^{(1)}\right]_{p_1} \\ A_{21} \leftarrow L_{21} = A_{21}/A_{11} \end{array}$$

Here $[x]_i$ indicates the ith element of vector x. It is important to realize that multiple partitionings of the same matrix reference the same data. Thus after swapping the elements of $A_{BR}^{(1)}$, A_{11} contains what was $[A_{BR}^{(1)}]_{p_1}$ before the swap.

6.4 Implementation

A FLAME implementation of the blocked algorithm in Figure 8 is given in Figure 9. Notice that a FLAME implementation of the unblocked algorithm

```
1  void FLA_LU( FLA_Obj A, FLA_Obj ipiv, int nb_alg )
2  {
3    < declarations >
4
5    FLA_Part_2x2( A,  &ATL, /**/ &ATR,
6                   /* ************** */
7                     &ABL, /**/ &ABR,
8                 /* with */ 0, /* by */ 0, /* submatrix */ FLA_TL );
9    FLA_Part_2x1( ipiv, &ipivT,
10                    /* ****** */
11                      &ipivB,
12                /* with */ 0, /* length submatrix */ FLA_TOP );
13
14   while (b = min( min( FLA_Obj_length( ABR ), FLA_Obj_width( ABR ) ), nb_alg ))
15   {
16     FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, /**/ &A01, &A02,
17                         /* ************* */    /* ******************** */
18                                /**/              &A10, /**/ &A11, &A12,
19                            ABL, /**/ ABR,         &A20, /**/ &A21, &A22,
20                /* with */ b, /* by */ b, /* A11 split from */ FLA_BR );
21     FLA_Repart_2x1_to_3x1( ipivT,                &ipiv0,
22                         /* ***** */             /* ***** */
23                                                  &ipiv1,
24                            ipivB,                &ipiv2,
25                /* with */ b, /* length ipiv1 split from */ FLA_BOTTOM );
26     FLA_Part_1x2( ABR,   &ABR_1, &ABR_2,
27                /* with */ b, /* width submatrix */ FLA_LEFT );
28     /* ********************************************************************** */
29
30     if ( nb_alg <= 4 ) FLA_LU_level2(ABR_1, ipiv1);
31     else               FLA_LU       (ABR_1, ipiv1, nb_alg/2);
32
33     FLA_Apply_pivots(FLA_SIDE_LEFT, FLA_NO_TRANSPOSE, ipiv1, ABL);
34     FLA_Apply_pivots(FLA_SIDE_LEFT, FLA_NO_TRANSPOSE, ipiv1, ABR_2);
35     FLA_Trsm(FLA_SIDE_LEFT, FLA_LOWER_TRIANGULAR,
36              FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
37              ONE, A11, A12);
38     FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A21, A12, ONE, A22);
39
40     /* ********************************************************************** */
41     FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,         A00, A01, /**/ A02,
42                                     /**/               A10, A11, /**/ A12,
43                               /* ************** */    /* ****************** */
44                                 &ABL, /**/ &ABR,         A20, A21, /**/ A22,
45                 /* with A11 added to submatrix */ FLA_TL );
46     FLA_Cont_with_3x1_to_2x1( &ipivT,                  ipiv0,
47                                                        ipiv1,
48                               /* ***** */             /* ***** */
49                                 &ipivB,                  ipiv2,
50                 /* with ipiv1 added to */ FLA_TOP );
51   }
52 }
```

Fig. 9. FLAME recursive LU factorization with partial pivoting.

would look similar. Let us assume that the latter is correctly implemented in the FLAME routine

```
void FLA_LU_level2(FLA_ObjA, FLA_Obj ipiv)
```

Now, the correctness of the algorithm in Figure 8 depends only on the correctness of the LU factorization with partial pivoting of $A_{BR}^{(1)}$ and the other operation. Thus, there is the option of implementing the LU factorization of the panel $A_{BR}^{(1)}$ as a recursive call to the given routine (line 31). Only when the panel becomes very small is a routine that uses level-2 BLAS (matrix-vector computations) called (line 30).

Notice that the implementation is very flexible in that it is neither purely recursive nor purely iterative. By playing with the algorithmic block size b (`nb_alg`), one can attain a purely recursive algorithm (when $b = n/2$ for an $m \times n$ input matrix A), purely iterative (by always calling `FLA_LU_level2` for the subproblem) or an iterative algorithm that recursively calls itself. An induction on the level of the recursion would establish the correctness of the given code. A more detailed discussion on the correctness of recursively formulated linear algebra algorithms can be found in Gustavson and Jonsson [2000] and Elmroth and Gustavson [2000].

7. EXPERIMENTS

In this section, we report the results of three different experiments. The first measures the impact of the FLAME approach on productivity. The second experiment demonstrates that FLAME makes the implementation of high-performance linear algebra algorithms more accessible to novices. In the final experiment we demonstrate that the attained performance is superb.

7.1 Productivity Experiment

As an experiment to measure, albeit roughly, the degree to which FLAME reduces code development time, one of the authors implemented all level-3 BLAS operations given in Figure 10 in terms of matrix-matrix multiplication. This exercise can easily require months to complete, even by a programmer who is experienced in the implementation of such operations. This includes time spent on extensive testing of the correctness of the implementations. The entire library of operations was completed using FLAME in a matter of about ten hours, including testing. As of this writing, we have used the resulting library for about nine months without encountering a bug in the implementations. The resulting code is included on the FLAME webpage given at the end of this paper. The prototype implementation of FLAME required to support the implementations of the level-3 BLAS took approximately one man-week.

It should be noted that the number of lines of code required for the implementation is not necessarily less than that required for a more conventional implementation. This is already evident when considering Figures 5 and 7. However, the effort is greatly reduced by the fact that the subroutines for the different operations use similar code skeletons. Moreover, we believe that the resulting code is substantially more readable.

SYMM	$C \leftarrow \alpha(L + \hat{L}^T)B + \beta C$		$C \leftarrow \alpha(U + \hat{U}^T)B + \beta C$	
	$C \leftarrow \alpha B(L + \hat{L}^T) + \beta C$		$C \leftarrow \alpha B(U + \hat{U}^T) + \beta C$	
SYRK	$\mathrm{lo}(C) \leftarrow \alpha\mathrm{lo}(AA^T) + \beta\mathrm{lo}(C)$		$\mathrm{up}(C) \leftarrow \alpha\mathrm{up}(AA^T) + \beta\mathrm{up}(C)$	
	$\mathrm{lo}(C) \leftarrow \alpha\mathrm{lo}(A^TA) + \beta\mathrm{lo}(C)$		$\mathrm{up}(C) \leftarrow \alpha\mathrm{up}(A^TA) + \beta\mathrm{up}(C)$	
SYR2K	$\mathrm{lo}(C) \leftarrow \alpha\mathrm{lo}(AB^T + BA^T) + \beta\mathrm{lo}(C)$		$\mathrm{up}(C) \leftarrow \alpha\mathrm{up}(AB^T + BA^T) + \beta\mathrm{up}(C)$	
	$\mathrm{lo}(C) \leftarrow \alpha\mathrm{lo}(A^TB + B^TA) + \beta\mathrm{lo}(C)$		$\mathrm{up}(C) \leftarrow \alpha\mathrm{up}(A^TB + B^TA) + \beta\mathrm{up}(C)$	
TRMM	$B \leftarrow \alpha LB$	$B \leftarrow \alpha L^TB$	$B \leftarrow \alpha UB$	$B \leftarrow \alpha U^TB$
	$B \leftarrow \alpha BL$	$B \leftarrow \alpha BL^T$	$B \leftarrow \alpha BU$	$B \leftarrow \alpha BU^T$
TRSM	$B \leftarrow \alpha L^{-1}B$	$B \leftarrow \alpha L^{-T}B$	$B \leftarrow \alpha U^{-1}B$	$B \leftarrow \alpha U^{-T}B$
	$B \leftarrow \alpha BL^{-1}$	$B \leftarrow \alpha BL^{-T}$	$B \leftarrow \alpha BU^{-1}$	$B \leftarrow \alpha BU^{-T}$

Fig. 10. Level-3 BLAS operations implemented as part of the productivity experiment.

7.2 Accessibility Experiment

It is our claim that the FLAME approach to the derivation and implementation of linear algebra algorithms greatly simplifies the development of linear algebra libraries. To demonstrate this, we handed a recipe for deriving algorithms, similar to the one in Section 4, to a class of computer science undergraduates at UT-Austin. These students had a limited background in linear algebra and essentially no background in high-performance computing. Using the FLAME approach they implemented blocked algorithms for linear algebra operations that are part of the level-3 BLAS. The results of this experiment can be found in Gunnels and van de Geijn [2001a].

7.3 Performance Experiment

To illustrate that correctness, simplicity, and modularity does not necessarily come at the expense of performance, we measured the performance of the LU factorization with pivoting given in Figure 9 followed by forward and backward substitution, that is, essentially the LINPACK benchmark. For comparison, we also measured the performance of the equivalent operations provided by ATLAS R3.2 [Whaley and Dongarra 1998].

Some details: Performance was measured on an Intel (R) Pentium (R) III processor-based laptop with a 256K L2 cache running the Linux (Red Hat 6.2) operating system. All computations were performed in 64-bit (double precision) arithmetic. For both implementations the same compiler options were used.

In Figure 11 we report performance for four different implementations, indicated by the curves marked.

ATLAS:. This curve reports performance for the LU factorization provided by ATLAS R3.2, using the BLAS provided by ATLAS R3.2.

ATL-FLAME:. This curve reports the performance of our LU factorization coded using FLAME with BLAS provided by ATLAS R3.2. The outer-most block size used for the LU factorization is 160 for these measurements. (Notice that multiples of 40 are optimal for the ATLAS matrix-matrix multiply on this architecture).

ITX-FLAME:. Same as the previous implementation, except that we provided our own optimized matrix-matrix multiply (ITXGEMM). Details of this optimization are the subject of another paper [Gunnels et al. 2001]. This time the outer-most block size was 128. (Notice that multiples of 64 are optimal for the ITXGEMM matrix-matrix multiplication routine on this architecture).

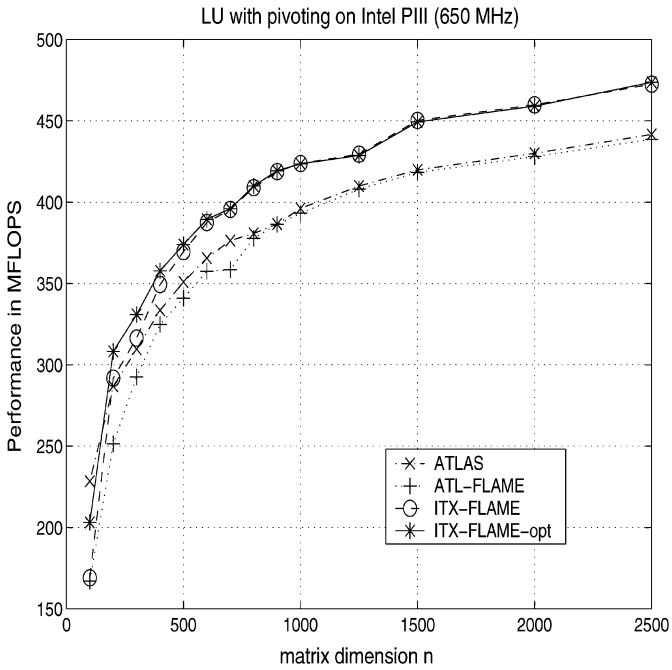


Fig. 11. Performance of LU factorization with pivoting followed by forward and backward substitution.

`ITX-FLAME-opt`:. Same as the ITX-FLAME implementation, except that we optimized the level-2 BLAS based LU factorization of an intermediate panel, as well as the pivot routine, by not using the high-level FLAME approach for those operations. For these routines we call `DSCAL`, `DGER`, and `DSWAP` directly.

For all implementations, the forward and backward substitutions are provided by the ATLAS R3.2 `DTRSV` routine.

Notice that for small matrices, the performance of `ATL-FLAME` is somewhat inferior to that of ATLAS, because of the overhead for manipulating the objects that encode the information about the matrices. This is due to the fact that this manipulation of objects introduces an $O(n)$ overhead which is amortized over a computational cost that is $O(n^3)$. When the level-2 BLAS based LU factorization is coded without this overhead, the performance is comparable for small matrices. The performance boost witnessed when the ITXGEMM matrix-matrix multiply kernel is used is entirely due to the superior performance of that kernel, relative to the ATLAS `DGEMM` implementation.

It is important to realize that the performance difference between the implementation offered as part of ATLAS R3.2 and our own implementation is not the point of this performance comparison or, more generally, of this paper. With some effort, either implementation can be improved to match the performance of the other. Our primary point is that FLAME enables one to expend markedly less time to implement these algorithms in a provably correct manner. At the same time, the resulting implementation attains performance comparable

to that of, what are widely considered to be, standard high-performance implementations.

8. FUTURE DIRECTIONS

Many aspects of the approach we have described are extremely systematic—the generation of the loop-invariants, the derivation of the algorithm, as well as the translation to code. Not discussed, is the fact that the analysis of the run-time of the resulting algorithm on sequential or, for that matter, parallel, architectures, is equally systematic. We are pursuing a project that exploits this systematic approach in order to *automatically* generate entire (parallel) linear algebra libraries as well as run-time estimates for the generated subroutines [Gunnels 2001]. The goal is to create a mechanism that will automatically choose between different algorithms based on architectural and/or problem parameters.

A considerably less ambitious project, already nearing completion, allows the user to program in a language-independent manner (i.e. by writing an ASCII version of the algorithms presented in this paper). Since it is our central thesis that the level of abstraction presented in this paper is the correct one, it seems an unnecessary onus to force the user to become familiar with the parameters and constraints of the underlying library. Obviously, the library must provide the necessary *functionality*, but the applications programmer should be concerned with nothing beyond the facilities provided by the library. Thus, the programmer should be allowed to express their algorithms at a higher level of abstraction, for example in terms of equations that can be automatically translated to (library) function calls.

9. CONCLUSION

A colleague of ours, Dr. Timothy Mattson of Intel, recently made the following observation: “Literature professors read literature. Computer Science professors should, at least occasionally, read code”. When one does this, certain patterns emerge and one tends to become more readily able to distinguish good code from bad.

In this paper, we have illustrated that a more formal approach to the design of matrix algorithms, combined with the right level of abstraction for coding, leads to a software architecture for linear algebra libraries that is dramatically different from the one that resulted from the more traditional approaches used by packages such as LINPACK, LAPACK, and ScaLAPACK. The approach is such that the library developer is forced to give careful attention to the derivation of the algorithm. The benefit is that the code produced is a direct translation of the resulting algorithm, greatly reducing opportunities for the introduction of common bugs related to indexing. Our experience shows that there is no significant loss of performance. Indeed, since more variants for a given operation can now be more easily developed, we often observe a performance benefit from the approach.

Throughout the paper we concentrate on the correctness of the algorithm. This is not the same as proving that the algorithm is numerically stable. While we do not claim that our methodology will automatically generate stable

algorithms, we do claim that the skeleton used to express the algorithm, and to implement the code, can be used to implement variants of several algorithms. These variants include those whose numerical stability properties are known. The methodology and framework also facilitate the discovery and implementation of new algorithms for which numerical properties can then be subsequently established and, thus, to select the variant whose properties most closely match the requirements of the application under consideration.

REFERENCES

AGARWAL, R., GUSTAVSON, F., AND ZUBAIR, M. 1994. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM J. Res. Dev. 38*, 5 (Sept.), 563–576.

ANDERSON, E., BAI, Z., DEMMEL, J., DONGARRA, J. E., DUCROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A. E., OSTROUCHOV, S., AND SORENSEN, D. 1992. *LAPACK Users' Guide.* SIAM, Philadelphia.

ANDERSEN, B. S., GUSTAVSON, F. G., AND WASNIEWSKI, J. 2000. A recursive formulation of Cholesky factorization of a matrix in packed storage. LAPACK Working Note 146 CS-00-441, University of Tennessee, Knoxville (May).

CHOI, J., DONGARRA, J. J., POZO, R., AND WALKER, D. W. 1992. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation.* IEEE Computer Society Press, 120–127.

CROUT, P. D. 1941. A short method for evaluating determinants and solving systems of linear equations with real or complex coefficients. *Trans AIEE 60*, 1235–1240.

DIJKSTRA, E. W. 2000. Under the spell of Leibniz's dream. Tech. Rep. EWD1298, The University of Texas at Austin (April). `http://www.cs.utexas.edu/users/EWD/`.

DONGARRA, J. J., BUNCH, J. R., MOLER, C. B., AND STEWART, G. W. 1979. *LINPACK Users' Guide* SIAM, Philadelphia.

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

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

DONGARRA, J. J., DUFF, I. S., SORENSEN, D. C., AND VAN DER VORST, H. A. 1991. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA.

DONGARRA, J. J., GUSTAVSON, F. G., AND KARP, A. 1984. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review 26*, 1 (Jan.), 91–112.

ELMROTH, E. AND GUSTAVSON, F. 2000. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM J. Res. Dev. 44*, 4, 605–624.

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

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

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

GUNNELS, J., LIN, C., MORROW, G., AND VAN DE GEIJN, R. 1998. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing* (*1998 IPPS/SPDP '98*). 110–116.

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

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

GUNTER, B. C., REILEY, W. C., AND VAN DE GEIJN, R. A. 2001. Parallel out-of-core cholesky and qr factorizations with pooclapack. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium* (*IPDPS*). IEEE Computer Society Press, Los Alamitos, CA.

GUSTAVSON, F. G. 1997. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. of Res. Dev. 41*, 6 (November), 737–755.

GUSTAVSON, F. G. 2001. New generalized matrix data structures lead to a variety of high-performance algorithms. In *The Architecture of Scientific Software*, R. F. Boisvert and P. T. P. Tang, Eds: Kluwer Academic Press, Orlando, FL.

GUSTAVSON, F., HENRIKSSON, A., JONSSON, I., KÅGSTRÖM, B., AND LING, P. 1998a. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. In *Applied Parallel Computing*, *Large Scale Scientific and Industrial Problems*, B. K. et al., Ed. Lecture Notes in Computer Science 1541. Springer-Verlag, New York, 195–206.

GUSTAVSON, F., HENRIKSSON, A., JONSSON, I., KÅGSTRÖM, B., AND LING, P. 1998b. Superscalar GEMM-based level 3 BLAS—the on-going evolution of a portable and high-performance library. In *Applied Parallel Computing*, *Large Scale Scientific and Industrial Problems*, B. K. et al., Ed. Lecture Notes in Computer Science 1541. Springer-Verlag, New York, 207–215.

GUSTAVSON, F. AND JONSSON, I. 2000. Minimal storage high-performance Cholesky factorization via blocking and recursion. *IBM J. Res. Dev. 44*, 6 (November), 823–850.

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

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

REILEY, W. C. 1999. Efficient parallel out-of-core implementation of the Cholesky factorization. Tech. Rep. CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin. (Dec.) Undergraduate Honors Thesis.

REILEY, W. C. AND VAN DE GEIJN, R. A. 1999. POOCLAPACK: Parallel Out-of-Core Linear Algebra Package. Tech. Rep. CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin (Nov.).

SMITH, B. T. ET AL. 1976. *Matrix Eigensystem Routines—EISPACK Guide*, Second ed. Lecture Notes in Computer Science 6. Springer-Verlag, New York.

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

STEWART, G. W. 1998. *Matrix Algorithms Volume 1*: *Basic Decompositions*. SIAM.

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

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

Received November 2000; revised November 2000 and June 2001; accepted September 2001

Formal Derivation of Algorithms: The Triangular Sylvester Equation

ENRIQUE S. QUINTANA-ORTÍ
Universidad Jaume I
and
ROBERT A. VAN DE GEIJN
The University of Texas at Austin

In this paper we apply a formal approach for the derivation of dense linear algebra algorithms to the triangular Sylvester equation. The result is a large family of provably correct algorithms. By using a coding style that reflects the algorithms as they are naturally presented, the correctness of the algorithms carries through to the correctness of the implementations. Analytically motivated heuristics are used to subsequently choose members from the family that can be expected to yield high performance. Finally, we report performance on the Intel (R) Pentium (R) III processor that is competitive with that of recursive algorithms reported previously in the literature for this operation.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: *Algorithm design and analysis; Efficiency; User Interfaces*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific achitectures*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

General Terms: Algorithms; Design; Theory; Performance

Additional Key Words and Phrases: Formal derivation; libraries, linear algebra; control theory; Sylvester equations

1. INTRODUCTION

In a recent paper the Formal Linear Algebra Methods Environment (FLAME) was introduced [Gunnels et al. 2001a]. FLAME is both a systematic approach for deriving (dense) linear algebra algorithms and a library for the implementation of the resulting algorithms. The rationale is that by formally deriving algorithms, correctness can be asserted. Moreover, by providing a framework for coding that mirrors the derived algorithms, the opportunity for the introduction

Primary support for this work came from the Visiting Researcher program of the Texas Institute for Computational and Applied Mathematics (TICAM).
Authors' addresses: E. S. Quintana-Ortí, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain; email: quintana@icc.uji.es; R. A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, 1 University Station C0500, Austin, TX 78712; email: rvdg@cs.utexas.edu.

of coding errors is greatly reduced and thus the correctness of the algorithms carries through to the implementations. In that paper the simple example of LU factorization was used to illustrate the basic techniques.

In this paper, we demonstrate the versatility of FLAME by concentrating on a more complex linear algebra operation, the solution of the Sylvester equation

$$AX + XB = C, \tag{1}$$

where A is an $m \times m$ matrix, B is $n \times n$, C and X are $m \times n$, and X is the sought-after solution. Let $\Lambda(A) = \{\alpha_i\}_{i=1}^{m}$ and $\Lambda(B) = \{\beta_j\}_{j=1}^{n}$ denote, respectively, the eigenspectra of A and B; then (1) has a (unique) solution if and only if $\alpha_i + \beta_j \neq 0$ for all $i = 1, \ldots, m$ and $j = 1, \ldots, n$. For further details on the existence of solutions of the Sylvester equation and numerical solvers see, for example, Bartels and Stewart [1972], Golub et al. [1979], and Hammarling [1982].

Sylvester equations have numerous applications in control theory, signal processing, filtering, image restoration, the decoupling of ordinary and partial differential equations, and block-diagonalization of matrices; see, for example, Aliev and Larin [1998], Calvetti and Reichel [1996], Golub and Van Loan [1996], and Sima [1996]. Also note that $B = A^T$ yields the Lyapunov equation such that everything derived here can be used (and simplified) for this type of equations, playing a vital role in many areas of computer-aided control system design.

In particular, we focus on the triangular case of Equation (1), where both A and B are (upper) triangular matrices. The solution of the triangular case arises, for example, as an intermediate subproblem in the Sylvester equation solver described in Bartels and Stewart [1972]. The cost of solving the triangular Sylvester equation of dimension $m \times n$, using a traditional serial (nonblocked) algorithm, is $m^2n + mn^2$ floating-point operations [Golub and Van Loan 1996].

While the solution of the triangular Sylvester equation is a well-studied problem, this paper presents a number of contributions:

—an illustration of the application of FLAME to a problem arising in control theory;
—the derivation of a large family of provably correct algorithms which includes, as a small subset, algorithms that are closely related to known traditional methods as well as recently proposed recursive algorithms;
—an analysis that provides heuristics for composing members of the family to yield high performance;
—a demonstration that high performance can be attained using the techniques here described.

Altogether, dozens of new high-performance algorithms and implementations are given.

While this paper is written to be self-contained, it is highly recommended that the reader consult the already mentioned earlier paper on FLAME as well as a recent paper that gives theoretical insight into high-performance matrix multiplication algorithms [Gunnels et al. 2001b]. Also, since the original submission of this paper we have refined the formal derivation approach. This more

refined approach is described in detail in Bientinesi et al. [2002]. In rewriting this paper, we have adopted the notation and steps of this latest paper.

This paper is structured as follows: In Section 2, we review traditional algorithms for the solution of the triangular Sylvester equation. A worksheet for the systematic derivation of linear algebra algorithms is discussed in Section 3. We use this worksheet to derive algorithms that are closely related to traditional algorithms as well as a more general family of algorithms in Section 4. In Section 5 we describe insights that we use to identify candidates from the family that are likely to yield high performance. Performance results on an Intel (R) Pentium (R) III processor are given in Section 6. A discussion of how the algorithms can be made practical is given in Section 7. A very brief discussion of experimental evidence of the numerical stability of the derived algorithms is given in Section 8. Concluding remarks follow in Section 9.

2. TRADITIONAL SOLVERS FOR THE TRIANGULAR SYLVESTER EQUATION

Blocked algorithms usually obtain a higher performance in modern computers by rearranging the computations as possible in terms of matrix multiplication [Dongarra et al. 1990]. The Linear Algebra Package (LAPACK) [Anderson et al. 1995] is a library that illustrates the benefits of reformulating algorithms to be rich in matrix-matrix products. Some of the latest research on high-performance implementation of matrix multiplication is embodied in the packages ATLAS [Whaley and Dongarrra 1998], PHIPAC [Bilmes et al. 1997], and ITXGEMM [Gunnels et al. 2001b].

In particular, blocked algorithms for solving the triangular Sylvester equation can easily be derived from the serial algorithms and are usually composed of two nested loops which iterate over blocks of columns and rows of the solution matrix. For each iteration of the inner loop a new block of the solution is obtained. Depending on the algorithm, some updates may be needed before a new block of the solution is obtained (leading to a lazy algorithm, which postpones much of the work) or after it is computed (an eager algorithm in such case).

As an example, we next present a traditional row-lazy/column-eager blocked triangular Sylvester equation solver. Assume A is partitioned into $b_m \times b_m$ blocks, $A_{i,j}$, $i, j = 1, \ldots, m/b_m$, and B is partitioned into $b_n \times b_n$ blocks, $B_{i,j}$, $i, j = 1, \ldots, n/b_n$. For simplicity, hereafter, we assume that m and n are integer multiples of b_m and b_n, respectively. These partitions induce conformal partitions of X and C into $b_m \times b_n$ blocks. Setting both b_m and b_n to 1 leads to element-wise algorithms, while setting only one of them produces row-oriented or column-oriented variants. The algorithm is stated in Figure 1, where we borrow the colon notation from MATLAB [The MathWorks, Inc. 1993]. This algorithm can easily be modified to overwrite C with the solution of the equation. The Sylvester equation arising at each iteration of the inner loop is usually solved using a nonblocked, row-oriented or column-oriented version of the algorithm. Notice that just before a new block of the solution is obtained, the corresponding block-row of C is updated with respect to the previous blocks of X in the same block-column, leading to a row-lazy updating scheme. On the other hand, when this new block is computed, it is used to update the remaining

```
for i = m/b_m : −1 : 1
   for j = 1 : n/b_n
      C_{i,j} = C_{i,j} − A_{i,i+1:m/b_m} X_{i,i+1:m/b_m ,j}
      solve A_{i,i} X_{i,j} + X_{i,j} B_{j,j} = C_{i,j}
      C_{i,j+1:n/n_b} = C_{i,j+1:n/n_b} − X_{i,j} B_{j,j+1:n/n_b}
   end
end
```

Fig. 1. Row-lazy/column-eager blocked triangular Sylvester equation solver.

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

Fig. 2. Worksheet for developing linear algebra algorithms.

blocks of C in the same block-row in a column-eager updating scheme. Three more variants of the algorithm are obtained by rearranging the updates to be row-lazy/eager and column-lazy/eager [Kågström and Poromaa 1992].

Recursive variants of these solvers have been recently developed in Jonsson and Kågström [2001]. Briefly, a recursive algorithm employs the same algorithm for solving the Sylvester equation in the inner loop, but uses a smaller dimension of the block sizes b_m and b_n. The higher efficiency of these algorithms is obtained by decoupling the dimensions of the blocks for the matrix multiplications from those of the Sylvester equations. The goal is to perform as much of the computation in terms of matrix multiplications as possible, while maximizing the size of the matrices involved in these products.

3. A WORKSHEET FOR DERIVING LINEAR ALGEBRA ALGORITHMS

In Figure 2, we give a generic "worksheet" for deriving a large class of linear algebra algorithms. Expressions in curly brackets denote predicates that describe the state of the various variables at the given point of the algorithm. For

this paper, it suffices to realize that the statements between the assertions in the curly brackets must be such that, at the indicated points in the algorithm, those assertions hold.

The generic linear algebra operation is given by $[D, E, F, \ldots] := \text{op}(A, B, C, D, \ldots)$. Notice that some operands may be both input and output parameters. Constraints on these parameters, for example, those that describe structure or original contents, are given by the predicate P_{pre}, the *precondition*. The *postcondition*, P_{post}, is the predicate that describes the desired state upon completion of the algorithm.

We will require that the state given by the predicate P_{inv}, the *loop-invariant*, be maintained at the top of the loop. Notice that P_{inv} must thus hold before the loop is entered, it must hold at the end of the loop so it will again hold at the top of the loop, and it will hold upon completion of the loop. This is indicated in Figure 2 at the various points where P_{inv} occurs in the assertions.

The *loop-guard*, G, is the condition under which the program remains in the loop. Thus, after the loop completes, $\neg G$ must hold. If $(P_{\text{inv}} \wedge \neg G) \Rightarrow P_{\text{post}}$ then we can conclude that the loop computes the desired result.

Since the loop-invariant must hold before the loop commences, the *initialization*, Step 4 in Figure 2, must have the property that starting in the state P_{pre}, the initialization leaves us in a state where P_{inv} holds.

In order to make progress toward the condition under which the loop is completed, we will see that regions of the operands to be used and/or updated must be identified and added to regions that have already been updated in a consistent manner. It is this identification of submatrices and shifting of boundaries that occurs in Steps 5a and 5b in Figure 2.

The exposure of submatrices to be used and/or updated dictates the state Q_{before} before any updates have actually occurred. The update itself, S_U, must be such that the state Q_{after} holds. This state must be such that after the shifting of the boundaries the loop-invariant again holds at the bottom of the loop.

In Section 4, we illustrate how the worksheet allows us to derive algorithms for the solution of Equation (1).

4. DERIVATION OF ALGORITHMS FOR THE TRIANGULAR SYLVESTER EQUATION

Let us use the notation $C := X = \Omega(A, B, C)$ to denote the operation that overwrites matrix C with the solution of Equation (1). Let $\hat{C}$ denote the original contents of C so that the precondition in Figure 2 becomes

$$C = \hat{C} \wedge \text{UpTriang}(A) \wedge \text{UpTriang}(B) \\ \wedge\, m(A) = n(A) = m(C) \wedge m(B) = n(B) = n(C)\,;$$

here, UpTriang(Z) returns *true* if Z is upper triangular, and $m(Z)$ and $n(Z)$ return, respectively, the row and column dimensions of Z. We now wish to compute

$$C = X = \Omega(A, B, \hat{C}), \tag{2}$$

the postcondition P_{post} in Figure 2.

Note: Throughout the rest of the paper, we will use the following notation:

$\hat{C}$	The original contents of matrix C
X	The solution of (1)
C	The current contents of C

First, we derive two block-row-oriented (with respect to matrix C) solvers by partitioning only the first of the coefficient matrices, A. Analogous block-column-oriented versions could be obtained by partitioning B instead of A. Finally, we will investigate algorithms where both A and B are partitioned into four quadrants.

4.1 Block-Row-Oriented Solvers

We start our derivation of block-row-oriented algorithms by partitioning matrix A into four quadrants

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} = 0 & A_{BR} \end{array} \right),$$

where A_{BR} is a $k_m \times k_m$ block. The indices $\{T\}$, $\{B\}$, $\{L\}$, and $\{R\}$ stand for top, bottom, left, and right, respectively. Accordingly, we next apply a conformal partition to C, $\hat{C}$, and X by blocks of rows

$$C \rightarrow \left(\begin{array}{c} C_T \\ \hline C_B \end{array} \right), \quad \hat{C} \rightarrow \left(\begin{array}{c} \hat{C}_T \\ \hline \hat{C}_B \end{array} \right), \quad \text{and} \quad X \rightarrow \left(\begin{array}{c} X_T \\ \hline X_B \end{array} \right),$$

where C_B, $\hat{C}_B$, and X_B are $k_m \times n$ blocks.

With these partitionings, Equation (1) can be rewritten as

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array} \right) \left(\begin{array}{c} X_T \\ \hline X_B \end{array} \right) + \left(\begin{array}{c} X_T \\ \hline X_B \end{array} \right) B = \left(\begin{array}{c} \hat{C}_T \\ \hline \hat{C}_B \end{array} \right).$$

Notice that the solution of this partitioned equation is given by

$$\left(\begin{array}{c} X_T \\ \hline X_B \end{array} \right) = \Omega\left(\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array} \right), B, \left(\begin{array}{c} \hat{C}_T \\ \hline \hat{C}_B \end{array} \right) \right) \tag{3}$$

$$= \left(\begin{array}{c} \Omega(A_{TL}, B, \hat{C}_T - A_{TR}\, \Omega(A_{BR}, B, \hat{C}_B)) \\ \hline \Omega(A_{BR}, B, \hat{C}_B) \end{array} \right). \tag{4}$$

To avoid the recomputation of an intermediate result, $X_B = \Omega(A_{BR}, B, \hat{C}_B)$, there are data dependencies which induce a strict order on the sequence of operations: provided C originally contains $\hat{C}$, first, $C_B := X_B = \Omega(A_{BR}, B, C_B)$ is solved, then the update $C_T := C_T - A_{TR}C_B$ is computed, and, finally, $C_T := X_T = \Omega(A_{TL}, B, C_T)$ is solved.

While Equation (4) gives all computations necessary to compute the solution, at an intermediate stage we would expect only some of these computations to have been computed. It is this insight that allows us to identify feasible loop-invariants, given in Table I. Notice that when any of these conditions is chosen

Table I. Feasible Loop-Invariants for the Block-Row-Oriented Algorithms to Solve the Triangular Sylvester Equation

Case	Loop-invariant	Resulting variant
R1	$\left(\begin{array}{c}C_T\\\hline C_B\end{array}\right) = \left(\begin{array}{c}\hat{C}_T\\\hline \Omega(A_{BR}, B, \hat{C}_B)\end{array}\right)$	Lazy
R2	$\left(\begin{array}{c}C_T\\\hline C_B\end{array}\right) = \left(\begin{array}{c}\hat{C}_T - A_{TR}\,\Omega(A_{BR}, B, \hat{C}_B)\\\hline \Omega(A_{BR}, B, \hat{C}_B)\end{array}\right)$	Eager

as the loop-invariant, P_{inv}, the loop-guard $G : m(C_T) > 0$ has the property that $(P_{\text{inv}} \wedge \neg G) \Rightarrow P_{\text{post}}$.

Now, the initialization (Step 4 in Figure 2)

$$A \rightarrow \left(\begin{array}{c|c}A_{TL} & A_{TR}\\\hline 0 & A_{BR}\end{array}\right), \quad C \rightarrow \left(\begin{array}{c}C_T\\\hline C_B\end{array}\right), \quad \text{and} \quad \hat{C} \rightarrow \left(\begin{array}{c}\hat{C}_T\\\hline \hat{C}_B\end{array}\right),$$

where A_{BR} is 0×0 and C_B and $\hat{C}_B$ have 0 rows, has the property that, starting in a state where the precondition holds, it leaves the matrix C in a state where P_{inv} is *true*.

Next, we must make progress toward making G *false*. In other words, we must derive a body of the loop that allows the computation of C to proceed forward (up) by b_m rows while ensuring that the loop-invariant is satisfied at the beginning of the next iteration. To derive these steps, we repartition matrices A, C, and $\hat{C}$ as

$$\left(\begin{array}{c|c}A_{TL} & A_{TR}\\\hline 0 & A_{BR}\end{array}\right) \rightarrow \left(\begin{array}{c|c|c}A_{00} & A_{01} & A_{02}\\\hline 0 & A_{11} & A_{12}\\\hline 0 & 0 & A_{22}\end{array}\right), \left(\begin{array}{c}C_T\\\hline C_B\end{array}\right) \rightarrow \left(\begin{array}{c}C_0\\\hline C_1\\\hline C_2\end{array}\right), \left(\begin{array}{c}\hat{C}_T\\\hline \hat{C}_B\end{array}\right) \rightarrow \left(\begin{array}{c}\hat{C}_0\\\hline \hat{C}_1\\\hline \hat{C}_2\end{array}\right),$$

where A_{11} is a $b_m \times b_m$ block and C_1 and $\hat{C}_1$ have b_m rows. The parameter b_m determines the granularity of our block-row-oriented algorithm. By setting $b_m = 1$ we obtain a non-blocked row-oriented algorithm.

The double lines in these partitionings indicate how far the computation has progressed. The idea now is that progress is made by shifting various submatrices, from one side of the double lines to the other. Thus, at the bottom of the loop, we will continue with

$$\left(\begin{array}{c|c}A_{TL} & A_{TR}\\\hline 0 & A_{BR}\end{array}\right) \leftarrow \left(\begin{array}{c|c|c}A_{00} & A_{01} & A_{02}\\\hline 0 & A_{11} & A_{12}\\\hline 0 & 0 & A_{22}\end{array}\right), \left(\begin{array}{c}C_T\\\hline C_B\end{array}\right) \leftarrow \left(\begin{array}{c}C_0\\\hline C_1\\\hline C_2\end{array}\right), \left(\begin{array}{c}\hat{C}_T\\\hline \hat{C}_B\end{array}\right) \leftarrow \left(\begin{array}{c}\hat{C}_0\\\hline \hat{C}_1\\\hline \hat{C}_2\end{array}\right).$$

4.1.1 *Lazy Algorithm.* If we wish to maintain loop-invariant R1,

$$\left(\begin{array}{c}C_T\\\hline C_B\end{array}\right) = \left(\begin{array}{c}\hat{C}_T\\\hline \Omega(A_{BR}, B, \hat{C}_B)\end{array}\right),$$

in terms of these repartitioned matrices, the current contents of C are given by

$$Q_{\text{before}} : \left(\begin{array}{c} C_0 \\ \hline\hline C_1 \\ \hline C_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} \hat{C}_0 \\ \hline \hat{C}_1 \end{array}\right) \\ \hline\hline \Omega(A_{22}, B, \hat{C}_2) \end{array}\right). \tag{5}$$

As part of the body of the loop, the contents of C must be updated so that

$$Q_{\text{after}} : \left(\begin{array}{c} C_0 \\ \hline\hline C_1 \\ \hline C_2 \end{array}\right) = \left(\begin{array}{c} \hat{C}_0 \\ \hline\hline \Omega\left(\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline 0 & A_{22} \end{array}\right), B, \left(\begin{array}{c} \hat{C}_1 \\ \hline \hat{C}_2 \end{array}\right)\right) \end{array}\right) \tag{6}$$

$$= \left(\begin{array}{c} \hat{C}_0 \\ \hline\hline \left(\begin{array}{c} \Omega(A_{11}, B, \hat{C}_1 - A_{12}\Omega(A_{22}, B, \hat{C}_2)) \\ \hline \Omega(A_{22}, B, \hat{C}_2) \end{array}\right) \end{array}\right). \tag{7}$$

The question now is what the statements, S_U, are that allow the computation to move forward while maintaining the indicated loop-invariant. Comparing Equations (5) and (7), we conclude that we need only to perform the operations

$$\begin{aligned} C_1 &:= C_1 - A_{12}C_2\,, \\ C_1 &:= \Omega(A_{11}, B, C_1)\,. \end{aligned}$$

The completed worksheet is stated in Figure 3. By recognizing that $\hat{C}$ is only introduced for the sake of the assertions in curly brackets, the final algorithm is given in Figure 4. We classify the resulting algorithm as "lazy," in the sense that at a given stage, as little as possible of C has been updated while allowing progress toward the solution to be made.

For those more comfortable with traditional algorithms, this is equivalent to the solver in Figure 5 (left). The lazy row-block oriented algorithms just presented, in the FLAME and the traditional formulations, are special cases of the algorithm in Figure 1, with $b_n = n$.

4.1.2 *Eager Algorithm.* If we wish to maintain the loop-invariant R2 in terms of the repartitioned matrices the current contents of C are instead given by

$$Q_{\text{before}} : \left(\begin{array}{c} C_0 \\ \hline\hline C_1 \\ \hline C_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} \hat{C}_0 - A_{02}\Omega(A_{22}, B, \hat{C}_2) \\ \hline \hat{C}_1 - A_{12}\Omega(A_{22}, B, \hat{C}_2) \end{array}\right) \\ \hline\hline \Omega(A_{22}, B, \hat{C}_2) \end{array}\right). \tag{8}$$

Now the contents of C must be updated so that

$$Q_{\text{after}} : \left(\begin{array}{c} C_0 \\ \hline\hline C_1 \\ \hline C_2 \end{array}\right) = \left(\begin{array}{c} \hat{C}_0 - (A_{01} | A_{02})\,\Omega\left(\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline 0 & A_{22} \end{array}\right), B, \left(\begin{array}{c} \hat{C}_1 \\ \hline \hat{C}_2 \end{array}\right)\right) \\ \hline\hline \Omega\left(\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline 0 & A_{22} \end{array}\right), B, \left(\begin{array}{c} \hat{C}_1 \\ \hline \hat{C}_2 \end{array}\right)\right) \end{array}\right) \tag{9}$$

$$= \left(\begin{array}{c} \hat{C}_0 - A_{01}\Omega(A_{11}, B, \hat{C}_1 - A_{12}\Omega(A_{22}, B, \hat{C}_2)) - A_{02}\Omega(A_{22}, B, \hat{C}_2) \\ \hline\hline \left(\begin{array}{c} \Omega(A_{11}, B, \hat{C}_1 - A_{12}\Omega(A_{22}, B, \hat{C}_2)) \\ \hline \Omega(A_{22}, B, \hat{C}_2) \end{array}\right) \end{array}\right). \tag{10}$$

Step	Annotated Algorithm: $C := \Omega(A, B, C)$
1a	$\left\{ \begin{array}{l} C = \hat{C} \ \wedge \ \mathrm{UpTriang}(A) \ \wedge \ \mathrm{UpTriang}(B) \\ \wedge \ m(A) = n(A) = m(C) \ \wedge \ m(B) = n(B) = n(C) \end{array} \right\}$
4	**Partition** $A \rightarrow \left(\begin{array}{c\|\|c} A_{TL} & A_{TR} \\ \hline\hline 0 & A_{BR} \end{array}\right), C \rightarrow \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right)$ **where** A_{BR} is 0×0 and C_B has 0 rows
2	$\left\{ \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) = \left(\begin{array}{c} \hat{C}_T \\ \hline\hline \Omega(A_{BR}, B, \hat{C}_B) \end{array}\right) \right\}$
3	**while** $m(C_T) > 0$ **do**
2,3	$\left\{ \left(\left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) = \left(\begin{array}{c} \hat{C}_T \\ \hline\hline \Omega(A_{BR}, B, \hat{C}_B) \end{array}\right) \right) \wedge (m(C_T) > 0) \right\}$
5a	**Determine block size** b **Repartition** $\left(\begin{array}{c\|\|c} A_{TL} & A_{TR} \\ \hline\hline 0 & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|\|c} A_{00} & A_{01} & A_{02} \\ \hline 0 & A_{11} & A_{12} \\ \hline\hline 0 & 0 & A_{22} \end{array}\right), \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline\hline C_2 \end{array}\right)$ **where** A_{11} is $b_m \times b_m$ and C_1 has b_m rows
6	$\left\{ \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline\hline C_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} \hat{C}_0 \\ \hline \hat{C}_1 \end{array}\right) \\ \hline\hline \Omega(A_{22}, B, \hat{C}_2) \end{array}\right) \right\}$
8	$C_1 := C_1 - A_{12}C_2$ $C_1 := \Omega(A_{11}, B, C_1)$
5b	**Continue with** $\left(\begin{array}{c\|\|c} A_{TL} & A_{TR} \\ \hline\hline 0 & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c\|\|c\|c} A_{00} & A_{01} & A_{02} \\ \hline\hline 0 & A_{11} & A_{12} \\ \hline 0 & 0 & A_{22} \end{array}\right), \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline\hline C_1 \\ \hline C_2 \end{array}\right)$
7	$\left\{ \left(\begin{array}{c} \hat{C}_0 \\ \hline\hline \left(\begin{array}{c} \Omega(A_{11}, B, \hat{C}_1 - A_{12}\Omega(A_{22}, B, \hat{C}_2) \\ \hline \Omega(A_{22}, B, \hat{C}_2)) \end{array}\right) \end{array} \right) \right\}$
2	$\left\{ \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) = \left(\begin{array}{c} \hat{C}_T \\ \hline\hline \Omega(A_{BR}, B, \hat{C}_B) \end{array}\right) \right\}$
	enddo
2,3	$\left\{ \left(\left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) = \left(\begin{array}{c} \hat{C}_T \\ \hline\hline \Omega(A_{BR}, B, \hat{C}_B) \end{array}\right) \right) \wedge \neg (m(C_T) > 0) \right\}$
1b	$\left\{ C = X = \Omega(A, B, \hat{C}) \right\}$

Fig. 3. Worksheet for the lazy block-row algorithm to solve the triagular Sylvester equation.

Comparing Equations (8) and (10), we conclude that we need only to perform the operations:

$$\begin{aligned} C_1 &:= \Omega(A_{11}, B, C_1)\,, \\ C_0 &:= C_0 - A_{01}C_1\,. \end{aligned}$$

The algorithm is stated in Figure 4 and is equivalent to the traditional solver in Figure 5 (right). We classify the resulting algorithm as "eager," in the sense that at a given stage, as much as possible of C has been updated without computing the final answer.

Partition $A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline 0 & A_{BR} \end{array}\right), C \rightarrow \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right)$

where A_{BR} is 0×0 and C_B has 0 rows

while $m(C_T) > 0$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline 0 & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline 0 & A_{11} & A_{12} \\ \hline\hline 0 & 0 & A_{22} \end{array}\right), \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline\hline C_2 \end{array}\right)$$

where A_{11} is $b_m \times b_m$ and C_1 has b_m rows

Lazy variant:	Eager variant:
$C_1 := C_1 - A_{12}C_2$	$C_1 := \Omega(A_{11}, B, C_1)$
$C_1 := \Omega(A_{11}, B, C_1)$	$C_0 := C_0 - A_{01}C_1$

Continue with

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline 0 & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline 0 & A_{11} & A_{12} \\ \hline 0 & 0 & A_{22} \end{array}\right), \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline\hline C_1 \\ \hline\hline C_2 \end{array}\right)$$

enddo

Fig. 4. Lazy and eager block-now-oriented triangular Sylvester equation solvers derived from R1 (lazy) and R2 (eager).

`for` $i = m/b_m : -1 : 1$

Lazy variant:	*Eager variant:*
$C_{i,:} = C_{i,:} - A_{i,i+1:m/b_m} C_{i+1:m/b_m,:}$	`solve` $A_{i,i}C_{i,:} + C_{i,:}B = C_{i,:}$
`solve` $A_{i,i}C_{i,:} + C_{i,:}B = C_{i,:}$	$C_{1:i-1,:} = C_{1:i-1,:} - A_{1:i-1,i}C_{i,:}$

`end`

Fig. 5. Lazy and eager traditional block-now-oriented triangular Sylvester equation solvers.

4.1.3 *Proving Correctness and Cost.* The following theorems prove the correctness of the the lazy and eager block-row-oriented algorithms and present their computational costs.

THEOREM 1. *The lazy and eager block-row-oriented algorithms in Figure 4 overwrite matrix C with the solution of the triangular Sylvester equation* $AX + XB = C$.

PROOF. For the lazy algorithm, the proof follows from the fact that the algorithm was derived so that the loop terminates and the assertions in Figure 3 are *true*. A similar worksheet can be given for the eager algorithm. □

THEOREM 2. *The lazy and eager block-row oriented triangular Sylvester equation solvers in Figure 4 both require* $m^2n + mn^2$ *floating-point operations.*

PROOF. We prove the theorem for the case where b_m is constant.

Table II. Cost of the Lazy and Eager Block-row-oriented Triangular Sylvester Equation Solvers Derived from R1 (Lazy) and R2 (Eager)

Operation	Cost	
	Lazy variant	Eager variant
$C_1 := C_1 - A_{12}C_2$	$\sum_{k=0}^{m/b_m-1} 2b_m k_m n \approx m^2 n$	
$C_1 := \Omega(A_{11}, B, C_1)$	$\sum_{k=0}^{m/b_m-1} (b_m^2 n + b_m n^2) \approx mn^2$	$\sum_{k=0}^{m/b_m-1} (b_m^2 n + b_m n^2) \approx mn^2$
$C_0 := C_0 - A_{01}C_1$		$\sum_{k=0}^{m/b_m-1} 2\bar{m} b_m n \approx m^2 n$
Total	$m^2 n + mn^2$	$m^2 n + mn^2$

In the algorithms in Figure 4 the size of C_B increases from $b_m \times n$ to $(m - b_m) \times n$, while the size of A_{BR} increases from $b_m \times b_m$ to $(m - b_m) \times (m - b_m)$. Assuming C_B currently is $k_m \times n$, and A_{BR} is thus currently $k_m \times k_m$, the different parts of the matrices have the following dimensions:

$$\begin{array}{c|c||c} \overbrace{A_{00}}^{\bar{m}} & \overbrace{A_{01}}^{b_m} & \overbrace{A_{02}}^{k_m} \\ \hline 0 & A_{11} & A_{12} \\ \hline\hline 0 & 0 & A_{22} \end{array} \begin{array}{l} \}\bar{m} \\ \}b_m \\ \}k_m \end{array} \qquad \begin{array}{c} \overbrace{C_0}^{n} \\ \hline C_1 \\ \hline\hline C_2 \end{array} \begin{array}{l} \}\bar{m} \\ \}b_m \\ \}k_m \end{array}$$

Here, $\bar{m} = m - k_m - b_m$.

The number of floating-point operations required to move the computation forward by b_m rows in the lazy and eager versions of the algorithm is given by

$$\begin{array}{ll} C_1 := C_1 - A_{12}C_2 & 2b_m k_m n \\ C_1 := \Omega(A_{11}, B, C_1) & b_m^2 n + b_m n^2 \\ C_0 := C_0 - A_{01}C_1 & 2\bar{m} b_m n \end{array}$$

For simplicity we neglect the lower-order terms in the computation of the cost of the algorithms. If we consider the algorithm to iterate for $k = 0, 1, 2, \ldots, m/b_m - 1$, then $k_m = kb_m$. Table II reports the cost of these three operations and the overall cost of the algorithms, proving the theorem. □

Notice that if the triangular Sylvester equations arising in the block-row-oriented algorithms are solved using a traditional, nonblocked solver, $m \approx n$, and $b_m \ll m$, half the computation is in operations involving smaller Sylvester equations and the other half is in matrix multiplications.

4.1.4 *Implementation.* The FLAME library allows code to mirror the derived algorithms, thus largely inheriting the proven correctness. Implementations of the lazy and eager algorithms using FLAME are given in Figure 6.

4.2 Block-Column-Oriented Solvers

By partitioning B instead of A, we obtain algorithms which compute the solution by column blocks.

4.3 A Family of Blocked Algorithms

We now show that a partitioning of both coefficient matrices leads to a family of 16 different blocked algorithms.

```
int FLA_Syl_Block_Row( int variant, FLA_Obj A, FLA_Obj B, FLA_Obj C, int bm_alg )
{
  // Declaration of local objects...

  FLA_Part_2x2( A,   &ATL, /**/ &ATR,
                     /* ********** */
                     &ABL, /**/ &ABR,
           /* with */ 0, /* by */ 0, /* submatrix */ FLA_BR );
  FLA_Part_2x1( C, &CT,
                  /**/
                  &CB,
           /* with length */ 0, /* submatrix */ FLA_BOTTOM );

  while ( FLA_Obj_length( CT ) != 0 ){
    bm = min( FLA_Obj_length( CT ), bm_alg );

    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00,  &A01,    /**/  &A02,
                                /**/            &A10,  &A11,    /**/  &A12,
                           /* ******** */       /* ******************** */
                           ABL, /**/ ABR,       &A20,  &A21,    /**/  &A22,
           /* with */ bm, /* by */ bm, /* A11 split from */ FLA_TL );
    FLA_Repart_2x1_to_3x1( CT,                  &C0,
                                                &C1,
                          /**/                  /**/
                           CB,                  &C2,
           /* with length */ bm, /* C1 split from */ FLA_TOP );
    /* ************************************************************ */

    if ( variant == LAZY )                                      /* C1 <- C1 - A12 C2 */
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A12, C2, ONE, C1 );

    FLA_Syl_level2( A11, B, C1 );                       /* C1 <- Omega( A11, B, C1 ) */

    if ( variant == EAGER )                                     /* C0 <- C0 - A01 C1 */
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A01, C1, ONE, C0 );

    /* ************************************************************ */
    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,      A00, /**/ A01,      A02,
                              /* ********** */     /* ****************** */
                                    /**/            A10, /**/ A11,      A12,
                              &ABL, /**/ &ABR,      A20, /**/ A21,      A22,
           /* with A11 added to submatrix */ FLA_BR );
    FLA_Cont_with_3x1_to_2x1( &CT,                  C0,
                              /**/                 /**/
                                                   C1,
                              &CB,                 C2,
           /* with C1 added to submatrix */ FLA_BOTTOM );
  }
}
```

Fig. 6. Block-row-oriented triangular Sylvester equation solvers implemented using FLAME.

Consider starting our derivation of blocked algorithms by partitioning both coefficient matrices into four quadrants

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right), \quad B \rightarrow \left(\begin{array}{c|c} B_{TL} & B_{TR} \\ \hline 0 & B_{BR} \end{array}\right),$$

where A_{BR} is a $k_m \times k_m$ block and B_{TL} is a $k_n \times k_n$ block. Accordingly, we next apply a conformal partition to C, $\hat{C}$, and X:

$$C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right), \quad \hat{C} \rightarrow \left(\begin{array}{c|c} \hat{C}_{TL} & \hat{C}_{TR} \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array}\right), \quad X \rightarrow \left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right),$$

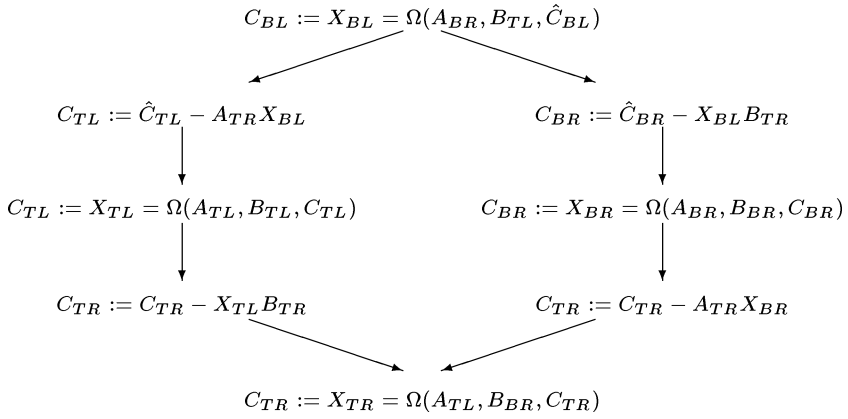


Fig. 7. Data dependencies for the partitioned triangular Sylvester matrix equation.

where C_{BL}, $\hat{C}_{BL}$, and X_{BL} are $k_m \times k_n$ blocks.

Now, Equation (1) becomes

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline 0 & A_{BR} \end{array}\right)\left(\begin{array}{c||c} X_{TL} & X_{TR} \\ \hline\hline X_{BL} & X_{BR} \end{array}\right) + \left(\begin{array}{c||c} X_{TL} & X_{TR} \\ \hline\hline X_{BL} & X_{BR} \end{array}\right)\left(\begin{array}{c||c} B_{TL} & B_{TR} \\ \hline\hline 0 & B_{BR} \end{array}\right) = \left(\begin{array}{c||c} \hat{C}_{TL} & \hat{C}_{TR} \\ \hline\hline \hat{C}_{BL} & \hat{C}_{BR} \end{array}\right)$$

and the solution, in partitioned form, is given by

$$\left(\begin{array}{c||c} X_{TL} & X_{TR} \\ \hline\hline X_{BL} & X_{BR} \end{array}\right) = \Omega\left(\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline 0 & A_{BR} \end{array}\right), \left(\begin{array}{c||c} B_{TL} & B_{TR} \\ \hline\hline 0 & B_{BR} \end{array}\right), \left(\begin{array}{c||c} \hat{C}_{TL} & \hat{C}_{TR} \\ \hline\hline \hat{C}_{BL} & \hat{C}_{BR} \end{array}\right)\right) \tag{11}$$

$$= \left(\begin{array}{c||c} \Omega(A_{TL}, B_{TL}, \hat{C}_{TL} - A_{TR}X_{BL}) & \Omega(A_{TL}, B_{BR}, \hat{C}_{TR} - A_{TR}X_{BR} - X_{TL}B_{TR}) \\ \hline\hline \Omega(A_{BR}, B_{TL}, \hat{C}_{BL}) & \Omega(A_{BR}, B_{BR}, \hat{C}_{BR} - X_{BL}B_{TR}) \end{array}\right). \tag{12}$$

The dependencies among the operations induce a certain order: the first operation that must be performed is solving $C_{BL} := X_{BL} = \Omega(A_{BR}, B_{TL}, \hat{C}_{BL})$; after that, only the updates $C_{TL} := \hat{C}_{TL} - A_{TR}X_{BL}$ or $C_{BR} := \hat{C}_{BR} - X_{BL}B_{TR}$ are possible, and so on. Figure 7 shows graphically these data dependencies. Due to the dependencies, there are only sixteen valid loop-invariants, as summarized in Table III. Notice that we label six of these cases as duals, since they involve analogous updates.

Repartition the matrices of the equation into nine blocks, as follows:

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

Table III. Valid Loop-Invariants for the Blocked Algorithms to Solve the Triangular Sylvester Equation

Case	Operations performed (Current contents of C)		Dual Case
C1	$\left(\begin{array}{c\|c} \hat{C}_{TL} & \hat{C}_{TR} \\ \hline X_{BL} & \hat{C}_{BR} \end{array}\right)$		
C2	$\left(\begin{array}{c\|c} \hat{C}_{TL} - A_{TR}X_{BL} & \hat{C}_{TR} \\ \hline X_{BL} & \hat{C}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} \hat{C}_{TL} & \hat{C}_{TR} \\ \hline X_{BL} & \hat{C}_{BR} - X_{BL}B_{TR} \end{array}\right)$	C11
C3	$\left(\begin{array}{c\|c} \hat{C}_{TL} - A_{TR}X_{BL} & \hat{C}_{TR} \\ \hline X_{BL} & \hat{C}_{BR} - X_{BL}B_{TR} \end{array}\right)$		
C4	$\left(\begin{array}{c\|c} X_{TL} & \hat{C}_{TR} \\ \hline X_{BL} & \hat{C}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} \hat{C}_{TL} & \hat{C}_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$	C12
C5	$\left(\begin{array}{c\|c} X_{TL} & \hat{C}_{TR} - X_{TL}B_{TR} \\ \hline X_{BL} & \hat{C}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} \hat{C}_{TL} & \hat{C}_{TR} - A_{TR}X_{BR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$	C13
C6	$\left(\begin{array}{c\|c} X_{TL} & \hat{C}_{TR} \\ \hline X_{BL} & \hat{C}_{BR} - X_{BL}B_{TR} \end{array}\right)$	$\left(\begin{array}{c\|c} \hat{C}_{TL} - A_{TR}X_{BL} & \hat{C}_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$	C14
C7	$\left(\begin{array}{c\|c} X_{TL} & \hat{C}_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$		
C8	$\left(\begin{array}{c\|c} X_{TL} & \hat{C}_{TR} - X_{TL}B_{TR} \\ \hline X_{BL} & \hat{C}_{BR} - X_{BL}B_{TR} \end{array}\right)$	$\left(\begin{array}{c\|c} \hat{C}_{TL} - A_{TR}X_{BL} & \hat{C}_{TR} - A_{TR}X_{BR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$	C15
C9	$\left(\begin{array}{c\|c} X_{TL} & \hat{C}_{TR} - A_{TR}X_{BR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} X_{TL} & \hat{C}_{TR} - X_{TL}B_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$	C16
C10	$\left(\begin{array}{c\|c} X_{TL} & \hat{C}_{TR} - A_{TR}X_{BR} - X_{TL}B_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$		

where A_{11} and B_{11} are, respectively, $b_m \times b_m$ and $b_n \times b_n$ blocks. Conformally, repartition

$$\left(\begin{array}{c\|\|c} C_{TL} & C_{TR} \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|\|c\|c} C_{00} & C_{01} & C_{02} \\ \hline\hline C_{10} & C_{11} & C_{12} \\ \hline\hline C_{20} & C_{21} & C_{22} \end{array}\right),$$

where C_{11} is a $b_m \times b_n$ block, and consider analogous repartitionings of $\hat{C}$ and X. Notice that, again, the double lines mark how far (the boundaries of) the computation has progressed. For the blocked algorithms, the solvers march through matrices C, $\hat{C}$, and X from the bottom-left corner to the top-right one, moving b_m rows and b_n columns at each iteration. In A, the double lines move toward the top-left corner, while in B the direction is toward the bottom-right corner. (If A and/or B were lower triangular matrices, all other possibilities in the direction of the computation would be observed.)

We next illustrate the derivation of a blocked algorithm resulting from a specific case, C2. In this case, the current contents of C are given by

$$Q_{\text{before}} : \left(\begin{array}{c\|\|c} \hat{C}_{TL} - A_{TR}X_{BL} & \hat{C}_{TR} \\ \hline\hline X_{BL} & \hat{C}_{BR} \end{array}\right) = \left(\begin{array}{c\|\|c} \left(\begin{array}{c\|c} \hat{C}_{00} - A_{02}X_{20} & \end{array}\right) & \left(\begin{array}{c\|c} \hat{C}_{01} & \hat{C}_{02} \end{array}\right) \\ \left(\begin{array}{c\|c} \hat{C}_{10} - A_{12}X_{20} & \end{array}\right) & \left(\begin{array}{c\|c} \hat{C}_{11} & \hat{C}_{12} \end{array}\right) \\ \hline\hline X_{20} & \left(\begin{array}{c\|c} \hat{C}_{21} & \hat{C}_{22} \end{array}\right) \end{array}\right).$$

Consider now the following repartitioning, which corresponds to the next stage where the computation has moved forward by a block of dimension $b_m \times b_n$:

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

$$\left(\begin{array}{c||c} C_{TL} & C_{TR} \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline\hline C_{20} & C_{21} & C_{22} \end{array}\right),$$

with $\hat{C}$ and X repartitioned as C. With this new repartitioning, we wish the contents of C to become

$$Q_{\text{after}} : \left(\begin{array}{c||c} C_{TL} - A_{TR}X_{BL} & C_{TR} \\ \hline\hline X_{BL} & C_{BR} \end{array}\right)$$

$$= \left(\begin{array}{c||c} \begin{array}{c|c} C_{00} - A_{02}X_{20} - A_{01}X_{10} & C_{01} - A_{02}X_{21} - A_{01}X_{11} \end{array} & C_{02} \\ \hline\hline \begin{array}{c|c} X_{10} & X_{11} \\ \hline X_{20} & X_{21} \end{array} & \begin{array}{c} C_{12} \\ \hline C_{22} \end{array} \end{array}\right).$$

Therefore, in order to move the computation forward in case C2 it can be easily shown that the operations shown in Figure 8 (center) must be performed (notice that $\hat{C}$ and X are only introduced for the sake of the assertions in curly brackets). We leave it as an exercise to the reader to derive the other variants.

THEOREM 3. *The algorithms in Figure 8 overwrite matrix C with the solution of the triangular Sylvester equation $AX + XB = C$.*

PROOF. Generating the worksheets for the blocked algorithms would yield a proof much like the one given for Theorem 1. □

The next theorem derives the cost of the blocked triangular Sylvester equation solver presented in Figure 8.

THEOREM 4. *Given that matrices X and C are $m \times n$, A is an $m \times m$ triangular matrix, and B is an $n \times n$ triangular matrix, the blocked triangular Sylvester equation solver in Figure 8 requires $m^2n + mn^2$ floating-point operations.*

PROOF. We prove the theorem for the case where the blocks sizes b_m and b_n are constant.

In the algorithm presented in Figure 8 the size of C_{BL} increases from $b_m \times b_n$ to $(m - b_m) \times (n - b_n)$, while the size of A_{BR} increases from $b_m \times b_m$ to $(m - b_m) \times (m - b_m)$, and that of B_{TL} increases from $b_n \times b_n$ to $(n - b_n) \times (n - b_n)$. Assuming C_{BL} is currently $k_m \times k_n$, and A_{BR}, B_{TL} are currently $k_m \times k_m$ and $k_n \times k_n$, respectively, the different parts of the matrices have the following dimensions:

$$\begin{array}{c|c||c} \overbrace{\bar{m}}^{} & \overbrace{b_m}^{} & \overbrace{k_m}^{} \\ \hline A_{00} & A_{01} & A_{02} \\ \hline\hline 0 & A_{11} & A_{12} \\ \hline 0 & 0 & A_{22} \end{array} \begin{array}{l} \}\bar{m} \\ \}b_m \\ \}k_m \end{array} \qquad \begin{array}{c|c||c} \overbrace{k_n}^{} & \overbrace{b_n}^{} & \overbrace{\bar{n}}^{} \\ \hline B_{00} & B_{01} & B_{02} \\ \hline\hline 0 & B_{11} & B_{12} \\ \hline 0 & 0 & B_{22} \end{array} \begin{array}{l} \}k_n \\ \}b_n \\ \}\bar{n} \end{array} \qquad \begin{array}{c|c||c} \overbrace{k_n}^{} & \overbrace{b_n}^{} & \overbrace{\bar{n}}^{} \\ \hline C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline\hline C_{20} & C_{21} & C_{22} \end{array} \begin{array}{l} \}\bar{m} \\ \}b_m \\ \}k_m \end{array}$$

Here, $\bar{m} = m - k_m - b_m$ and $\bar{n} = n - k_n - b_n$.

ALGORITHM 1. $C \leftarrow X$, *where* $AX + XB = C$
(Blocked)

Partition

$$A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline 0 & A_{BR} \end{array}\right), \quad B \rightarrow \left(\begin{array}{c||c} B_{TL} & B_{TR} \\ \hline\hline 0 & B_{BR} \end{array}\right), \quad C \rightarrow \left(\begin{array}{c||c} C_{TL} & C_{TR} \\ \hline\hline C_{BL} & C_{BR} \end{array}\right),$$

where A_{BR}, B_{TL}, **and** C_{BL} **are** 0×0

while C_{TR} **is not** 0×0 **do**

Determine block sizes b_m *and* b_n

Repartition

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

$$\left(\begin{array}{c||c} C_{TL} & C_{TR} \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} C_{00} & C_{01} & C_{02} \\ \hline\hline C_{10} & C_{11} & C_{12} \\ \hline\hline C_{20} & C_{21} & C_{22} \end{array}\right),$$

where A_{11} **is** $b_m \times b_m$, B_{11} **is** $b_n \times b_n$, **and** C_{11} **is** $b_m \times b_n$

C1 variant:	*C2 variant:*	*C3 variant:*
$C_{10} := C_{10} - A_{12}C_{20}$		
$C_{10} := \Omega(A_{11}, B_{00}, C_{10})$	$C_{10} := \Omega(A_{11}, B_{00}, C_{10})$	$C_{10} := \Omega(A_{11}, B_{00}, C_{10})$
$C_{11} := C_{11} - C_{10}B_{01}$	$C_{11} := C_{11} - C_{10}B_{01}$	$C_{11} := C_{11} - C_{10}B_{01}$
$C_{21} := C_{21} - C_{20}B_{01}$	$C_{21} := C_{21} - C_{20}B_{01}$	$C_{21} := \Omega(A_{22}, B_{11}, C_{21})$
$C_{21} := \Omega(A_{22}, B_{11}, C_{21})$	$C_{21} := \Omega(A_{22}, B_{11}, C_{21})$	
$C_{11} := C_{11} - A_{12}C_{21}$	$C_{11} := C_{11} - A_{12}C_{21}$	$C_{11} := C_{11} - A_{12}C_{21}$
$C_{11} := \Omega(A_{11}, B_{11}, C_{11})$	$C_{11} := \Omega(A_{11}, B_{11}, C_{11})$	$C_{11} := \Omega(A_{11}, B_{11}, C_{11})$
	$C_{00} := C_{00} - A_{01}C_{10}$	$C_{00} \leftarrow C_{00} - A_{01}C_{10}$
	$C_{01} := C_{01} - A_{01}C_{11}$	$C_{01} \leftarrow C_{01} - A_{01}C_{11}$
	$C_{01} := C_{01} - A_{02}C_{21}$	$C_{01} \leftarrow C_{01} - A_{02}C_{21}$
		$C_{12} \leftarrow C_{12} - C_{10}B_{02}$
		$C_{12} \leftarrow C_{12} - C_{11}B_{12}$
		$C_{22} \leftarrow C_{22} - C_{21}B_{12}$

Continue with

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

$$\left(\begin{array}{c||c} C_{TL} & C_{TR} \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} C_{00} & C_{01} & C_{02} \\ \hline\hline C_{10} & C_{11} & C_{12} \\ \hline\hline C_{20} & C_{21} & C_{22} \end{array}\right)$$

enddo

Fig. 8. Blocked triangular Sylvester equation solvers derived from C1, C2, and C3.

The number of floating point operations required to move the computation forward is given by

$C_{10} := \Omega(A_{11}, B_{00}, C_{10})$	$b_m^2 k_n + b_m k_n^2$
$C_{21} := \Omega(A_{22}, B_{11}, C_{21} - C_{20}B_{01})$	$2b_n k_m k_n + b_n^2 k_m + b_n k_m^2$
$C_{11} := \Omega(A_{11}, B_{11}, C_{11} - C_{10}B_{01} - A_{12}C_{21})$	$2b_m b_n k_n + 2b_m b_n k_m + b_m^2 b_n + b_m b_n^2$
$C_{00} := C_{00} - A_{01}C_{10}$	$2b_m k_n \bar{m}$
$C_{01} := C_{01} - A_{01}C_{11} - A_{02}C_{21}$	$2b_m b_n \bar{m} + 2b_n k_m \bar{m}$

For simplicity we neglect the lower-order terms in the computation of the cost of the algorithm, which leads us to consider only the operations denoted as $\Omega(X_{10})$, $\bar{C}_{21}$, $\Omega(X_{21})$, $\bar{C}_{00}$, and $\bar{C}_{01}$. If we consider the algorithm to iterate for $k = 0, 1, 2, \ldots, \max(m/b_m, n/b_n) - 1$, then $k_m = kb_m$ and $k_n = kb_n$, and some algebra proves the theorem. □

As we would hope, the blocked algorithm requires the same computational cost as the block-row solvers. The blocked algorithms for the remaining 15 cases are obtained by simply deriving the set of operations that will satisfy the loop-invariant in each case, and they all can be shown to present the same cost.

5. HEURISTICS

In Section 4, we derived a large number of algorithms for the solution of the triangular Sylvester equation. The question now becomes how to design a near-optimal implementation. In this section, we present both theoretical and practical insights that help guide the way.

First, let us review observations regarding blocked algorithms in general. All are designed to spend a substantial part of the computation in the matrix multiplication (DGEMM) kernel. Thus, it makes sense to pick block sizes b_m and b_n that allow the individual calls to the matrix-matrix multiply to attain the highest performance, subject to other constraints.

Consider the matrix multiplication $C \leftarrow AB + C$, where A is $m \times k$, B is $k \times n$, and then C is $m \times n$. In Gunnels et al. [2001b], it was shown that, for architectures with two levels of cache memory, there are two block sizes that influence the performance of DGEMM: b_1 and b_2, which are related to the sizes of the L1 and L2 caches, respectively. In Table IV we show how the three matrix dimensions, m, n, and k, affect performance of DGEMM. Where it says "large" in the table, the larger the dimensions, the better the performance.

Logic suggests that we attempt to minimize the amount of computation in the solution of the smaller Sylvester equations that show up in the body of the loop, thereby maximizing the amount of computation in DGEMM at this level (notice that each of these smaller Sylvester equations can be solved using the same algorithm, generating a recursion with multiple levels). In Section 4.1 we mentioned that the block-row-oriented algorithms spend approximately half of the computation in these subproblems. By symmetry, the same is true for the block-column-oriented algorithms. We now show that, if the block sizes are chosen carefully, the blocked algorithms spend only a third of the computation in the subproblems.

Consider the part of the total computational cost that is spent in solving the Sylvester equations in the body of the loop if we apply, for example, the algorithm derived for C2, with $b_m, b_n \ll m, n$. (Here we can assume without loss of generality that $m/b_m \geq n/b_n$.) At each iteration of the algorithm we need to solve two large Sylvester equations, for $\Omega(X_{10})$ and $\Omega(X_{21})$, with an overall computational cost of $mn^2 + \frac{b_m n^3}{3b_n}\left(\frac{bm}{bn} - 2\right)$ flops (floating-point arithmetic operations). The question therefore becomes what are the values of b_m and b_n that minimize this value, that is, $\min_{\{b_m, b_n\}} \frac{b_m}{b_n}\left(\frac{bm}{bn} - 2\right)$. This minimum is attained for $b_m = b_n$ and, in such case, approximately a third of the computation is spent in the subproblems. We can conclude that we should pursue the use of square block sizes.

Let us revisit the observations made regarding the matrix-matrix multiplication. Table V gives two heuristics for choosing different blocked and/or block-row and block-column when targeting different levels of the memory hierarchy.

Table IV. Factors Affecting the Performance of the Matrix Multiplication

Shape	Performance
Two dimensions large, one dimension equal to b_2	Best
One dimension large, two dimensions equal to b_2	↑
Two dimensions equal to b_2, one dimension equal to b_1	↓
One dimension equal to b_2, two dimensions equal to b_1	Good
All dimensions $\leq b_1$	Worse
One or more dimensions equal to 1	Worst

Table V. Two Heuristics for Choosing Algorithms and Block Sizes

Case		Strategy		Comment:
m	n	Heuristic 1	Heuristic 2	dims. passed to DGEMM
large	large	blocked ($b_m = b_n = b_2$)	block-row ($b = b_2$)	Two large/one equal to b_2.
b_2	large	block-column ($b = b_2$)		One large/two equal to b_2.
large	b_2	block-row ($b = b_2$)		One large/two equal to b_2.
b_2	b_2	blocked ($b_m = b_n = b_1$)	block-row ($b = b_1$)	Two equal to b_2/one equal to b_1.
b_1	b_2	block-column ($b = b_1$)		One equal to b_2/two equal to b_1.
b_1	b_1	any		Blocking is less important.

Consider the top-level blocking. If one takes $b_m = b_n = b_2$ (as in Heuristic 1) then two-thirds of the computation will be in matrix multiplication. If, on the other hand, one considers a block-row-oriented algorithm, with block size $b_m = b_2$ (as in Heuristic 2), then only half of the computation is in terms of the matrix-matrix multiplication. However, in this second alternative, one of the dimensions involved in the matrix-matrix multiplication always equals b_2, one always equals n, and the third ranges from small to large (C2 acts as an eager algorithm). A similar observation can be made for a block-column-oriented algorithm. By contrast, if a blocked algorithm is used, one dimension always equals b_2 while two of the dimensions range from small to large or vice-versa. Furthermore, when blocked algorithms are used, a larger number of calls to DGEMM are made. Thus, *in practice*, we can expect the calls to DGEMM employed by the block-row or block-column algorithms to attain higher performance than the calls employed by the blocked algorithms when $b_m = b_n = b_2$.

Heuristic 2 does not require blocked algorithms. However, we now show that by picking b_m and b_n carefully, some blocked algorithms can be used to implement the second heuristic in a particularly elegant fashion. As indicated in Table VI, notice that by setting $b_m = m$ or $b_n = n$, some of the cases of the blocked algorithms become either block-column or block-row algorithms, respectively. Notice that four variants, C1, C2, C3, and C11, have the property that by picking the block sizes carefully, they can become either block-row or block-column algorithms. For these variants, the following strategy will automatically generate an algorithm which conforms to the second heuristic: recursively call the given algorithm with, progressively, the block sizes $b_2 \times n$, $b_2 \times b_2$, $b_1 \times b_2$, $b_1 \times b_1$, followed by some strategy for solving the small $b_1 \times b_1$ Sylvester equation subproblems that remain at the lowest level of the recursion. For example, we can employ the same algorithm to reduce the subproblem to a certain size, $b_0 \times n$, apply an additional level of recursion to reduce it further to $b_0 \times b_0$, and solve this square subproblems using a nonblocked algorithm.

Table VI. Illustration That by Picking the Block Sizes Appropriately, Some of the Blocked Algorithms Become Block-Row or Block-Column Algorithms

Resulting algorithm	Cases	
	$b_m = m$	$b_n = n$
Lazy block-column	**C1**, **C2**, C4, C5	—
Eager block-column	**C3**, C6, C8, **C11**	—
Lazy block-row	—	**C1**, **C11**, C12, C13
Eager block-row	—	**C2**, **C3**, C14, C15

6. EXPERIMENTAL RESULTS

In this section we report the performance attained by our algorithms as the rate of computations achieved in millions of flops per second (MFLOPS/sec). We consider here triangular Sylvester equations of dimension $m \times n$, and an operation count of $m^2n + mn^2$ flops.

We report performance on an Intel (R) Pentium (R) III (650-MHz) processor with a 16-kbyte L1 data cache and a 256-kbyte L2 cache running RedHat Linux 7.1. All computations were performed in 64-bit (double precision) arithmetic, and the same options were used when compiling the different implementations. For all experiments, the implementations were linked to an implementation of the Basic Linear Algebra Subprograms (BLAS) [Lawson et al. 1979; Dongarra et al. 1988; Dongarra et al. 1990] provided by ATLAS (Release R3.2). This library provides high-performance implementations of commonly encountered matrix operations. For some experiments, the matrix multiplication kernel (DGEMM) provided by ATLAS was replaced by our own high-performance implementation, ITXGEMM [Gunnels et al. 2001b], while the rest of the BLAS were provided by ATLAS.

We analyze performance for five different implementations, indicated by the curves marked as follows:

—`unb`. Eager column-oriented implementation using Fortran-77. (We also implemented eager/lazy and row-/column-oriented variants, but the results were inferior for those.) Due to the poor performance of this approach, we only report results for the smaller problem sizes.

—`Trad. blocked`. Row-eager/column-eager blocked implementation of the solver using using Fortran-77, similar to that in Figure 1. (We also implemented all other variants; their performance proved inferior.)

—`C1, C2, and C3`. Our implementations of the solvers using FLAME and the heuristics described in Section 5. At the lowest level of the recursion, the subproblems are computed using the unblocked (`unb`) solver. Although we implemented all blocked, block-row, and block-column algorithms, we only present results for C1, C2, and C3. The remaining blocked algorithms obtained performance that was virtually identical to C1, C2, and C3. Also, we already argued that the block-row and block-column algorithms are special cases of C1, C2, C3, and C11.

Figure 9 (left) reports the performance for these algorithms using DGEMM from ATLAS R3.2. For reference, we also include the performance of the DGEMM

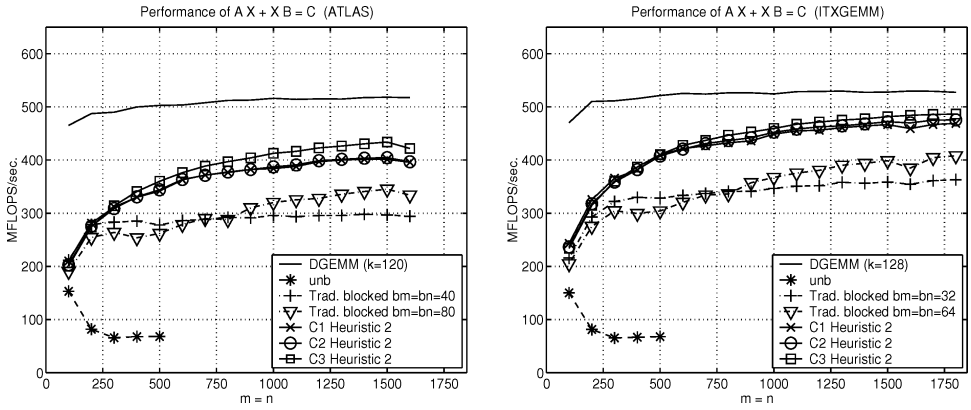


Fig. 9. Performance of our triangular Sylvester equation solvers using DGEMM from ATLAS R3.2 (Left) and ITXGEMM (Right).

routine for a matrix multiplication with $k = b_2$. We only report the results using the second heuristic described in Section 5, with block sizes $120 \times n$, 120×120, 40×120, 40×40, 20×40, and 20×20, for the different levels of the recursion. It is interesting to note that algorithm C3 performs somewhat better than C1 and C2.

Experts in the field will appreciate the fact that C3 performs most of its computation in the updates $C_{00} := C_{00} - A_{01}C_{10}$ and $C_{22} := C_{22} - C_{21}B_{12}$, which are both rank-$k$ updates. Typical implementations of DGEMM are tuned to perform best for that particular case of matrix multiplication. By contrast, C1 and C2 perform some or most of their computation in updates that involve matrices with small m or n dimensions, the performance of which is often not as highly tuned.

In Figure 9 (right) we report performance from the same experiments as reported in Figure 9 using DGEMM from ITXGEMM instead. For ITXGEMM, block sizes $128 \times n$, 128×128, 64×128, 64×64, 16×64, and 16×16, for the different levels of the recursion, yield high performance. Since this implementation of matrix multiplication is highly tuned for all matrix sizes, differences in performance of C1, C2, and C3 are less noticable. In addition, a higher percentage of the performance of matrix multiplication is achieved.

Figure 10 reports the performance of the best solver (that derived from C3) using both Heuristic 1 and Heuristic 2. It appears that on this architecture the algorithms benefit noticeably from the larger matrix sizes involved in the calls to DGEMM in Heuristic 2.

Although not reported here, our solvers, when linked to ATLAS R3.2, obtain performance similar to that reported in Jonsson and Kågström [2001] when we adjust for the different clock rate of the processor on which they performed their experiments.

7. APPLICATION TO PRACTICAL IMPLEMENTATIONS

This paper is intended to demonstrate the application of the FLAME approach to a more complex operation. However, we now show how the insights gained

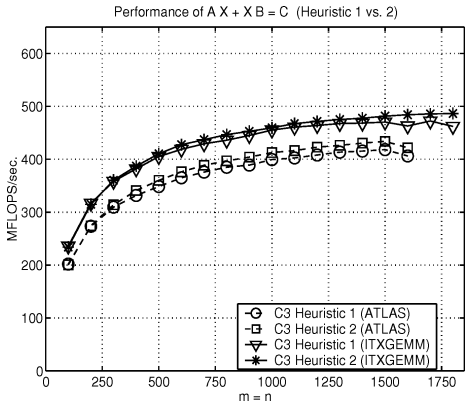


Fig. 10. Performance of the triangular Sylvester equation solvers using Heuristic 1 and Heuristic 2.

in the simpler setting where matrices A and B are triangular matrices can be easily extended to the more complex case that arises in practice.

7.1 Solution of the General Sylvester Equation

Let us start by considering the general case as it typically occurs in practice, for example in control theory:

$$\bar{A}\bar{X} + \bar{X}\bar{B} = \bar{C}, \tag{13}$$

where $\bar{A}$ and $\bar{B}$ are general, real valued, square matrices. A typical first step is to compute real Schur decompositions of matrices $\bar{A}$ and $\bar{B}$:

$$\bar{A} \rightarrow Q_A A Q_A^T \quad \text{and} \quad \bar{B} \rightarrow Q_B B Q_B^T, \tag{14}$$

where Q_A and Q_B are unitary matrices and A and B are *quasi* upper triangular (block triangular with 1×1 and 2×2 blocks on the diagonal). Then

$$(Q_A^T \bar{A} Q_A)(Q_A^T \bar{X} Q_B) + (Q_A^T \bar{X} Q_B) Q_B^T \bar{B} Q_B = (Q_A^T \bar{C} Q_B)$$

or

$$AX + XB = C, \tag{15}$$

where $X = Q_A^T \bar{X} Q_B$ and $C = Q_A^T \bar{C} Q_B$. Thus, the steps for computing the solution to Equation (13) become:

(1) Compute the real Schur decompositions. Approximate cost: $30(m^3 + n^3)$ flops[1].
(2) Compute $C = Q_A^T \bar{C} Q_B$. Approximate cost: $2m^3 + 2n^3$ flops.
(3) Solve Equation (15). Approximate cost: $m^2 n + m n^2$ flops.
(4) Compute $\bar{X} = Q_A X Q_B^T$. Approximate cost: $2m^3 + 2n^3$ flops.

[1]Part of this computation is an iteration that converges to the decomposition. Since the rate of convergence depends on the spectrum of the matrices, this operation count varies depending on the input matrices.

This then produces the desired solution of the original problem stated in Equation (13).

The first observation is that the cost of the solution of the triangular Sylvester equation is only a small part of the overall cost.

7.2 Solution of the Quasitriangular Sylvester Equation

The second observation is that what is really needed is a family of algorithms for the solution of the quasitriangular Sylvester equation. We now discuss how the algorithms presented in the previous sections must be adjusted for this case.

(1) *A is at most 2×2 and B is 1×1*: In this case Equation (15) is equivalent to $(A + \beta I)x = c$, where $B = \beta$ is a scalar and x and c are vectors. Since A is at most 2×2, this equation can be solved via Gaussian elimination with partial pivoting.

(2) *A is at most 2×2 and B is 2×2*: In this case Equation (15) is given by

$$A(x_1 \;\; x_2) + (x_1 \;\; x_2) \begin{pmatrix} \beta_{11} & \beta_{12} \\ \beta_{21} & \beta_{22} \end{pmatrix} = (c_1 \;\; c_2),$$

where β_{ij} is a scalar and x_j and c_j are vectors. This problem can be restated as

$$\left(\begin{array}{c|c} A + \beta_{11} I & \beta_{21} I \\ \hline \beta_{12} I & A + \beta_{22} I \end{array} \right) = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}.$$

Since this system is at most 4×4, we can again use Gaussian elimination with partial pivoting.

(3) *A is upper triangular and B is 1×1*: In this case Equation (15) is again equivalent to $(A + \beta I)x = c$. Since A is upper triangular, so is $A + \beta I$ and therefore any algorithm for the solution of a triangular system can be used.

(4) *A is quasitriangular and B is 1×1*: For this case, we can apply any of the algorithms derived in Section 4.1 with the constraint that the block size b_m should always be chosen so that A_{11} is either upper triangular or 2×2. If A_{11} is 2×2, the subproblem $C_{11} := \Omega(A_{11}, B, C_{11})$ can then be computed using the approach in case 1. If it is upper triangular, the approach in case 3 above can be employed.

(5) *A is quasitriangular and B is 2×2*: For this case, we can apply any of the algorithms derived in Section 4.1 with the constraint that the block size b_m should always be chosen so that A_{11} is either 1×1 or 2×2. The subproblem $C_{11} := \Omega(A_{11}, B, C_{11})$ can then be computed using the approach in case 2 above.

(6) *A and B are both quasitriangular*: If both A and B are quasitriangular, then any of the blocked algorithms discussed in previous sections can be applied provided the following:
   —The block sizes b_m and/or b_n are chosen so that A_{10}, A_{20}, A_{21}, B_{10}, B_{20}, and B_{21} all only contain zeroes.
   —Eventually an algorithm is called for the smaller subproblems with 1×1 or 2×2 matrix B_{11}.

With this, we have arrived at a practical implementation for the quasitriangular Sylvester equation.

8. STABILITY EXPERIMENTS

The stability analysis of unblocked algorithms for the triangular Sylvester equation has been well studied; see, for example, the thorough treatment in Higham [2002]. In this section, we discuss experimental data that provides some evidence that the derived blocked algorithms exhibit stability similar to that of the unblocked algorithms. Additional comments about the need for systematic stability analyses are given in the conclusion of the paper.

All stability experiments in this section were run on an Intel (R) Pentium III platform using MATLAB 6.0 (machine precision $\varepsilon \approx 2.2 \times 10^{-16}$). Implementations of all the codes were developed for that purpose using a special (MATLAB) version of the FLAME Application Programming Interface (API).

In order to evaluate the numerical properties of our solvers, we use an example of a generalized Sylvester equation

$$\hat{A}X\hat{B} + \hat{C}X\hat{D} = \hat{E}, \tag{16}$$

from Gardiner et al. [1992]. In that example

$$\begin{aligned} \hat{A} &= \mathrm{diag}(1, 2, \ldots, m) + U_m, \\ \hat{B} &= I_n + 2^{-p}U_n^T, \\ \hat{C} &= I_m + 2^{-p}U_m^T, \\ \hat{D} &= 2^{-p}I_n - \mathrm{diag}(n, n-1, \ldots, 1) + U_m, \end{aligned}$$

where p is a parameter and U_k is a $k \times k$ matrix with unit entries below the diagonal and all other entries zero. As p is increased, the system approaches singularity and the numerical condition of Equation (16) gets worse. The problem is then transformed into a standard Sylvester Equation (1) by solving the systems $A = \hat{C}^{-1}\hat{A}$ and $B = \hat{D}^T\hat{B}^{-T}$. The right-hand-side matrix C is set to be an $m \times n$ matrix so that the true solution matrix X has all unit entries. Matrices A and B are then reduced to real Schur form using the QR algorithm, resulting in quasitriangular matrices with 1×1 and 2×2 diagonal blocks. The right-hand-side matrix C is updated with the corresponding orthogonal matrices computed by the QR algorithm.

Table VII reports the relative error

$$\|X - \tilde{X}\|_\infty / \|X\|_\infty$$

of the solution $\tilde{X}$ computed by means of a traditional nonblocked solver (`unb`) and our variants C1 and C3 for a problem of size $m = n = 128$. The remaining blocked algorithms obtained numerical performances very similar to those of C1 and C3. At the lowest level, all algorithms dealt with tiny Sylvester equations of size 1×1, 1×2, 2×1, and 2×2, transforming these into an equivalent small linear system, and using Gaussian elimination with partial pivoting to solve these small subproblems. Our blocked variants employed block sizes $b_m \times b_n = 64\times32$, 32×16, and 16×8 at the first three levels of the recursion. At deeper levels b_m and b_n were set to 1 or 2, depending on whether a 2×2 block is encountered in the

Table VII. Numerical Performance of the Sylvester Solvers

	$\|X - \hat{X}\|_\infty / \|\hat{X}\|_\infty$		
p	unb	C1	C3
10	1.296496677377634e-10	2.250667595002365e-10	2.256006397319266e-10
20	2.336569168070665e-07	2.336517769523788e-07	2.336504013166624e-07
30	8.702811282624316e-04	8.702811330852456e-04	8.702811100522159e-04
40	8.466351601860348e-01	8.466396448984768e-01	8.466546182582525e-01

diagonal of the quasitriangular matrices A and/or B. No significant differences were found when we used nonsquare block sizes, smaller block sizes at deeper levels of the recursion, etc.

The conclusion of our experiments is that it appears that the derived blocked algorithms exhibit numerical stability that is similar to that of the unblocked algorithms.

9. CONCLUDING REMARKS

In this paper, we have made a number of contributions to the solution of the triangular Sylvester equation. These include:

—The systematic derivation and proof of correctness of a family of algorithms using the FLAME approach.

—The implementation of the family using the FLAME library.

—A heuristic for composing high-performance implementations from members of the family of algorithms.

—A demonstration of excellent performance.

—Altogether, we have presented dozens of new algorithms for the solution of this problem.

Many of our observations can be extended to blocked algorithms for dense linear algebra operations in general. These include:

—The FLAME approach is a powerful tool for the derivation of provably correct blocked algorithms.

—The FLAME library provides a prototype environment for the rapid implementation of such algorithms.

—The heuristics developed for composing members of the family are likely to be extendable to blocked algorithms in general.

—The FLAME approach and library naturally support the hybrid recursive/iterative algorithms that are both general and required for near-optimal implementation.

Clearly, this paper raises as many questions as it solves. For example, given that a large number of algorithms are systematically derived, the question of the numerical stability of the resulting algorithms must be addressed. While for some operations, for the triangular Sylvester equation whether or not all algorithms are equally stable is not immediately obvious. Thus, a systematic approach to deriving stability results would be highly desirable. We intend to study this question in the future.

ACKNOWLEDGMENTS

We thank John Gunnels for his comments on a preliminary version of this paper. We are particularly grateful to G. W. (Pete) Stewart for his valuable feedback on the original manuscript.

ADDITIONAL INFORMATION

For additional information on FLAME please visit

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

REFERENCES

ALIEV, F. AND LARIN, V. 1998. *Optimization of Linear Control Systems: Analytical Methods and Computational Algorithms*. Stability and Control: Theory, Methods and Applications, vol. 8. Gordon and Breach, New York, NY.

ANDERSON, E., BAI, Z., BISCHOF, C., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., OSTROUCHOV, S., AND SORENSEN, D. 1995. *LAPACK Users' Guide*, 2nd ed. SIAM, Philadelphia, PA.

BARTELS, R. AND STEWART, G. 1972. Solution of the matrix equation $AX + XB = C$: Algorithm 432. *Comm. ACM 15*, 9, 820–826.

BIENTINESI, P., GUNNELS, J. A., MYERS, M. E., QUINTANA-ORÍ, E. S., AND VAN DE GEIJN, R. A. 2002. The science of deriving dense linear algebra algorithms. Tech. Rep. CS-TR-02-53, Department of Computer Sciences, The University of Texas at Austin, Austin, TX. Available online at `http://www.cs.utexas.edu/users/flame/`.

BILMES, J., ASANOVIC, K., CHIN, C., AND DEMMEL, J. 1997. Optimizing matrix multiply using PHiPAC: a portable, high-performace, ANSI C coding methodology. In *Proceedings of the 11th International Conference on SuperComputing* (ICS'97)(July 1997), ACM Press, New York, NY.

CALVETTI, D. AND REICHEL, L. 1996. Application of ADI iterative methods to the restoration of noisy images. *SIAM J. Matrix Anal. Appl. 17*, 1, 165–186.

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

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

GARDINER, J., LAUB, A., AMATO, J., AND MOLER, C. 1992. Solution of the Sylvester matrix equation $A \times B_T + C \times D_T = E$. *ACM Trans. Math. Softw. 18*, 2, 223–231.

GOLUB, G. AND VAN LOAN, C. 1996. *Matrix Computations*, 3rd ed. Johns Hopkins University Press, Baltimore, MD.

GOLUB, G. H., NASH, S., AND VAN LOAN, C. F. 1979. A Hessenberg–Schur method for the problem $AX + XB = C$. *IEEE Trans. Automat. Control AC-24*, 6, 909–913.

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

GUNNELS, J., HENRY, G., AND VAN DE GEIJN, R. 2001b. A family of high-performance matrix multiplication algorithms. In *Computational Science—ICCS 2001, Part I*, V. Alexander, J. Dongarra, B. Julianno, R. Renner, and C. Kenneth Tan, Eds. Lecture Notes in Computer Science, vol. 2073. Springer-Verlag, New York, NY, 51–60.

HAMMARLING, S. J. 1982. Numerical solution of the stable, non-negative definite Lyapunov equation. *IMA J. Numer. Anal. 2*, 303–323.

HIGHAM, N. J. 2002. *Accuracy and Stability of Numerical Algorithms*, 2nd ed. SIAM, Philadelphia, PA.

JONSSON, I. AND KÅGSTRÖM, B. 2001. Recursive blocked algorithms for solving triangular matrix equations—part I: one-sided and coupled Sylvester equations. Department of Computing Science and HPC2N Report UMINF-01.05. Umeå University, Umeå, Sweden.

KÅGSTRÖM, B. AND POROMAA, P. 1992. Distributed and shared memory block algorithms for the triangular Sylvester equation with sep^{-1} estimator. *SIAM J. Matrix Anal. Appl. 13,* 1, 90–101.

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

SIMA, V. 1996. *Algorithms for Linear-Quadratic Optimization*. Pure and Applied Mathematics, vol. 200. Marcel Dekker. New York, NY.

THE MATHWORKS, INC. 1993. *The MATLAB Control Toolbox, Version 3.0b*. The MathWorks, Inc., Natick, MA.

WHALEY, R. C. AND DONGARRRA, J. 1998. Automatically tuned linear algebra software. In *Proc. of SC'98*(Nov. 1998). IEEE Computer Society Press, Los Alamitos, CA.

Received September 2001; revised October 2002; accepted February 2003

The Science of Deriving Dense Linear Algebra Algorithms

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

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

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

General Terms: Algorithms; Design; Theory; Performance

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

Dedicated to the Memory of Edsger Wybe Dijkstra.

This research was partially sponsored by NSF grants ACI-0203685 and ACI-0305163.
Authors' addresses: P. Bientinesi, M. E. Myers, and R. A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: {pauldj;myers,rvdg}@cs.utexas.edu. J. A. Gunnels, IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598; email: gunnels@us.ibm.com; E. S. Quintana-Ortí, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain; email: quintana@icc.uji.es.

1. INTRODUCTION

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

1.1 Verification and Derivation

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

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

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

1.2 Derivation of Linear Algebra Algorithms

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

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

1.3 Relation to Other Articles on Our Project

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

—The first article [Gunnels et al. 2001] gave a broad outline of the approach, introducing the concept of formal derivation and its application to dense linear algebra algorithms. In that article we also showed that by introducing an Application Programming Interface (API) for coding the provably correct algorithms, claims about the correctness of the algorithms allow claims about the correctness of the implementation to be made. Finally, we showed that excellent performance can be attained by employing these methods. The primary vehicle for illustrating the techniques in that article was the LU factorization.

  The treatment of the systematic approach was relatively vague in that article in part because we had not yet had the insight that a "worksheet", introduced later in this article, provides a convenient framework for the derivation of the algorithms.

—We showed that the method applies to more complex operations in the second article [Quintana-Ortí and van de Geijn 2003]. In that article we showed how a large number of new high-performance algorithms for the solution of the triangular Sylvester equation can be derived using the methodology.

  As originally submitted, that article did not include the worksheet. However, this (third) was written and submitted before the final submission of the second paper, which was subsequently rewritten by explaining the derivation of those algorithms with the aid of the worksheet.

In a number of workshop articles we have also given a more cursory treatment of the techniques [Gunnels and van de Geijn 2001b; Bientinesi et al. 2002].

This, third article focuses specifically on the system for deriving algorithms: it gives a step-by-step "recipe" that novice and veteran alike can use to rapidly derive correct algorithms. The second article discussed above should be viewed as an application of the current one to a much more complex operation.

1.4 Scope

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

1.5 Organization of the Article

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

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

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

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

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

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

2. CORRECTNESS OF ALGORITHMS

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

2.1 Notation

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

$$\alpha := 1,$$

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

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

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

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

EXAMPLE The predicate

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

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

2.2 The Correctness of Loops

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

[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.

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

3.1 A Typical Linear Algebra Operation

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

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

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

$$L(x_1 \mid x_2 \mid \cdots \mid x_n) = (b_1 \mid b_2 \mid \cdots \mid b_n)$$

or

$$(Lx_1 \mid Lx_2 \mid \cdots \mid Lx_n) = (b_1 \mid b_2 \mid \cdots \mid b_n).$$

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

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

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

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

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

solution X,

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

is given in Figure 1.

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*.

3.2 An Algorithm, Annotated for Verification

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

Figure 2) is given by

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

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

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

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

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

where (relating this to Figure 1) B_L and $\hat{B}_L$ represent the first $j - 1$ columns of B and $\hat{B}$, respectively. (Notice that subscripts L and R stand for Left and Right, respectively.) Thus, at the top of the loop the desired current contents of B are given by $P_{\text{inv}} : (B_L \parallel B_R) = (L^{-1}\hat{B}_L \parallel \hat{B}_R) \wedge \ldots$, the loop-invariant (Step 2). Since the loop in Figure 1 is executed as long as not all columns have been updated, the loop-guard is given by $\text{n}(B_L) \neq \text{n}(B)$ (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 Figure 2. In Figure 1, the left-most column of the set of columns yet to be updated is updated, moving it to the set of columns that have been updated. In our notation, we accomplish this by repartitioning as in Step 5a, which means that the current contents of B, in terms of the repartitioned matrices, is given by

$$P_{\text{before}} : (B_0 \parallel b_1 \mid B_2) = (L^{-1}\hat{B}_0 \parallel \hat{b}_1 \mid \hat{B}_2) \wedge \ldots$$

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

$$P_{\text{after}} : (B_0 \parallel b_1 \mid B_2) = (L^{-1}\hat{B}_0 \parallel L^{-1}\hat{b}_1 \mid \hat{B}_2) \wedge \ldots$$

(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 Figure 2 are *true*, proving that the algorithm is correct. Finally, we notice that $\hat{B}$ was only introduced for the benefit of the assertions in Figure 2. Since the update in the body of the loop never referenced $\hat{B}$ or its submatrices, a final algorithm is given in Figure 3.

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

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

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

$$
\begin{array}{l}
\textbf{Partition} \quad B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array} \right) \\
\quad \textbf{where} \quad \mathrm{n}(B_L) = 0 \\
\textbf{while} \quad G : (\mathrm{n}(B_L) \neq \mathrm{n}(B)) \quad \textbf{do} \\
\quad \textbf{Repartition} \\
\qquad \left(\begin{array}{c|c} B_L & B_R \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} B_0 & b_1 & B_2 \end{array} \right) \\
\qquad\quad \textbf{where} \quad \mathrm{n}(b_1) = 1 \\
\quad b_1 := L^{-1} b_1 \\
\quad \textbf{Continue with} \\
\qquad \left(\begin{array}{c|c} B_L & B_R \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} B_0 & b_1 & B_2 \end{array} \right) \\
\textbf{enddo}
\end{array}
$$

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

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

Fig. 4. Worksheet for developing linear algebra algorithms.

4. A SYSTEM FOR DERIVING LINEAR ALGEBRA ALGORITHMS

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

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

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

4.1 Step 1: Precondition and Postcondition

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

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

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 $B := L^{-1}B = \mathrm{TRSM}(L, B)$, where L is a nonsingular $m \times m$ lower triangular matrix and B is an $m \times n$ general matrix. For the matrix multiplication on the right to be well-defined, the column dimension of L must match the row dimension of B. We will want to overwrite B with the result without requiring a work array.

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

EXAMPLE (CONTINUED) The variables for TRSM can be described by the precondition

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

where, as before, $\hat{B}$ indicates the original contents of B. For brevity, typically we will only explicitly state the most important part of this predicate: $P_{\mathrm{pre}} : (B = \hat{B}) \wedge \ldots$.

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 $P_{\mathrm{post}} : B = L^{-1}\hat{B}$.

4.2 Step 2: Deriving Loop-Invariants

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

EXAMPLE (CONTINUED) Let us pick variable L. Since it is triangular, we partition it as

$$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right).$$

Here L_{TL} is square so that both submatrices on the diagonal are themselves lower triangular. (The subscripts TL, BL, and BR stand for Top-Left, Bottom-Left, and Bottom-Right, respectively.)

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

EXAMPLE (CONTINUED) Substituting the partitioning of L into the postcondition yields

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

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

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

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

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

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

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

Table I. Possible Loop-Invariants for the TRSM Example When the Process is Started by Partitioning Matrix L into Quadrants. The Reason Listed for Rejecting the Loop-Invariant Given in the Column Labeled "Comment" May Not be the Only Reason for Doing So

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

EXAMPLE (CONTINUED)

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

and hence

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

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.

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 the subset of possible loop-invariants given in Table I.

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

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

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

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

$$\left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) = \left(\begin{array}{c} L_{TL}^{-1}\hat{B}_T \\ \hline \hat{B}_B \end{array}\right) \tag{7}$$

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

4.3 Step 3: Derive the Loop-Guard

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

EXAMPLE (CONTINUED) Comparing the loop-invariant in (7) with the postcondition $B = L^{-1}\hat{B}$ we see that *if* $B = B_T$, $\hat{B} = \hat{B}_T$, *and* $L = L_{TL}$ then the loop-invariant implies the postcondition: 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 $G : \big(\text{m}(L_{TL}) \neq \text{m}(L)\big)$ meets this condition.

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

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

4.4 Step 4: Derive the Initialization

The loop-invariant P_{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:

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

$$\textbf{where}\ \ B_T \text{ and } \hat{B}_T \text{ have 0 rows and } L_{TL} \text{ is } 0 \times 0$$

in Step 4 in Figure 5. Since then B_T and $\hat{B}_T$ have no rows, and $B_B = B$ and $\hat{B}_B = \hat{B}$, it is not hard to see that $\{P_{\text{pre}}\}S_I\{P_{\text{inv}}\}$ 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* 2 *in Step* 2.

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

Fig. 5. Annotated algorithm for TRSM example.

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

4.5 Step 5: Determine How to Make Progress

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

EXAMPLE (CONTINUED) Loop-guard G indicates that eventually L_{TL} should equal all of L, at which point G becomes *false* and the loop is exited. After the initialization, L_{TL} is 0×0. The partitioning of L is also such that L_{TL} should always be square. Thus, the repartitioning should be such that as the computation proceeds, the dimensions of L_{BR} should decrease as the dimensions of L_{TL} increase. This is accomplished by the shifting of the double-lines as indicated in Steps 5a and 5b in Figure 5.

Notice that we are exposing **blocks** of rows and/or columns as part of the movement of the double lines. The reason for this is related to performance and will become more clearly apparent in Appendix A.2.

4.6 Step 6: Derive the State in Terms of the Repartitioned Variables

The repartitioning of the variables and the loop-invariant P_{inv} in Step 5a dictates P_{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 P_{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 Figure 5 identify that

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

Textual substitution into the loop-invariant yields the state

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

4.7 Step 7: Derive the State in Terms of the Repartitioned Variables at the Bottom of the Loop Body

The redefinition via partitioning of the variables in Step 5b and the loop-invariant P_{inv} dictate the desired state of the variables after the update S_U

and after the shifting of the double-lines, P_{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 Figure 5 identifies the following equivalent submatrices:

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

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

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

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

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

4.8 Step 8: Derive How Submatrices Must be Updated

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

EXAMPLE (CONTINUED) Comparing (8) and (9) we find that the updates

$$B_1 := B_1 - L_{10}B_0$$
$$B_1 := L_{11}^{-1}B_1$$

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

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

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

Partition $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$ and $L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$

where B_T has 0 rows and L_{TL} is 0×0

while $\mathrm{m}(L_{TL}) \neq \mathrm{m}(L)$ **do**

Determine block size b

Repartition

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

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

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

enddo

Fig. 6. Algorithm for the TRSM example.

4.9 Step 9: The Final Algorithm

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

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

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

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

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

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

$$B \rightarrow (B_L \parallel B_R).$$

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

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

and

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

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

If one repartitions

$$(B_L \parallel B_R) \rightarrow (B_0 \parallel b_1 \mid B_2) \ldots,$$

one recovers exactly those algorithms, while the repartitioning

$$(B_L \parallel B_R) \rightarrow (B_0 \parallel B_1 \mid B_2) \ldots,$$

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.

4.10 Recursion

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

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

5. CONCLUSIONS AND FUTURE DIRECTIONS

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

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

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

—If it were possible to fully *automate* the derivation and implementation of provably correct algorithms for linear algebra operations, then one could claim that this area of research is well-understood.

A prototype system, implemented by Sergey Kolos at UT-Austin as part of a semester project, automatically derives all algorithms for some linear algebra operations using Mathematica [Wolfram 1996] as a tool. Similarly, a semi-automatic tool has been developed by one of the authors that uses Mathematica to perform many of the indicated steps for a very broad class of operations. These prototype tools demonstrate that automation may be achievable.

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

Some preliminary work on the automatic derivation of cost analyses for parallel architectures shows that this may be possible [Gunnels 2001].

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

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

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

Additional Information

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

APPENDIX

A. PRACTICAL CONSIDERATIONS

This article 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 article, 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 article, we show in Figure 7 how an appropriately defined API, our FLAME/C library [Gunnels and van de Geijn 2001a, 2001b; Gunnels et al. 2001; Bientinesi et al. 2005b], can be used to program algorithms so that the code closely resembles the algorithms. A similar API has been defined for MATLAB (FLAME@lab) [Moler et al. 1987; Bientinesi et al. 2005b]. Finally, the Parallel Linear Algebra Package (PLAPACK) [van de Geijn 1997; Alpatov et al. 1997] API has been extended to allow parallel implementations to also more closely reflect the presented algorithms.

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

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

A.2 Experimental Results

In this section we illustrate how the derivation methodology, combined with the FLAME/C API, leads to high-performance algorithms and implementations for the TRSM operation. Performance was measured on a 650 MHz Intel (R) Pentium (R) III processor-based laptop with a 256K L2 cache running the Linux (Red Hat 7.1) operating system. All computations were performed in 64-bit (double precision) arithmetic. For our implementations, the FLAME/C API linked to BLAS provided by the ATLAS Version R3.2 BLAS library for the Pentium III processor [Whaley and Dongarra 1998]. In other words, whenever a call like

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

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

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

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

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

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

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

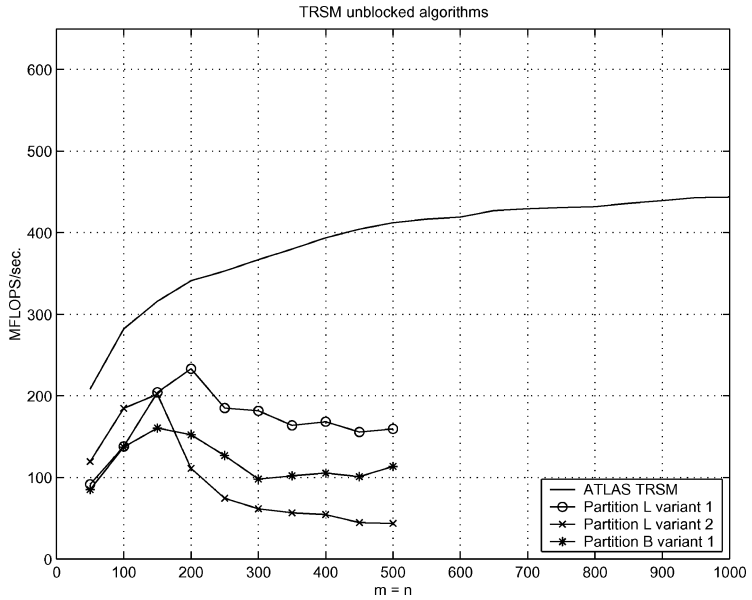


Fig. 8. Performance of unblocked TRSM implementations.

`FLA_Ger` is made, it results in a call to the corresponding BLAS routine, in this case the rank-1 update `dger`. The only exception occurs when `FLA_Gemm` is called: For some of the experiments, the ATLAS implementation of the DGEMM routine is called by this routine. For other experiments, our ITXGEMM [Gunnels et al. 2001] implementation of DGEMM is called instead.

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

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

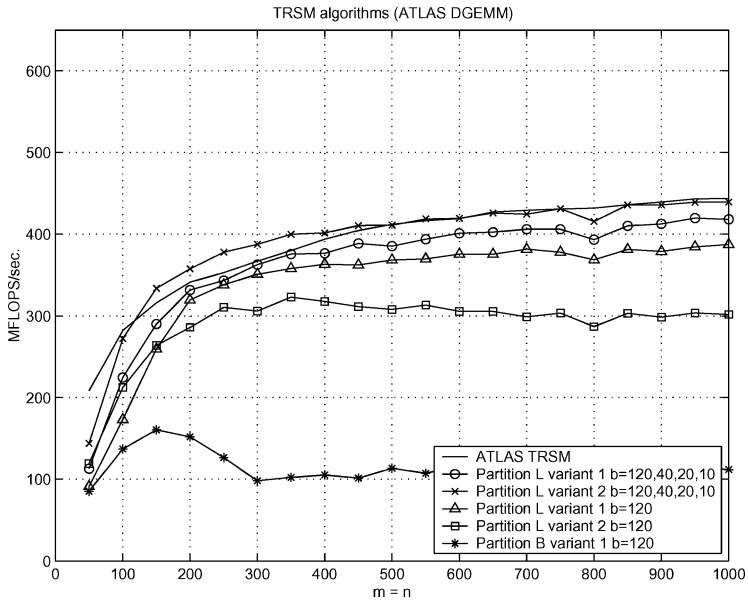


Fig. 9. Performance of blocked and recursive TRSM implementations. Here FLAME/C is interfaced with the DGEMM provided by ATLAS.

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

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

A.3 How to Choose

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

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

Often, implementations of `dgemm` achieve better performance when k is small relative to m and n than when m is small relative to n and k. When considering the update $B_1 := B_1 - L_{10}B_0$ in Variant 1, the block size, b, is typically chosen

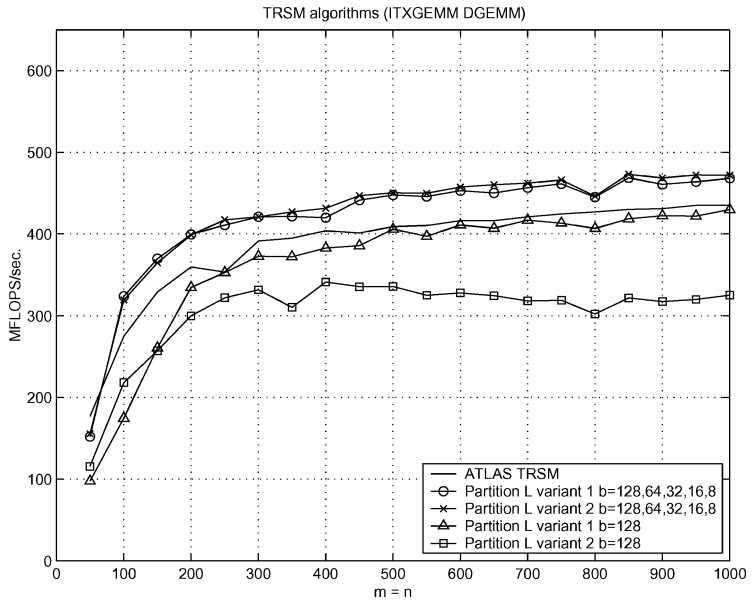


Fig. 10. Performance of blocked and recursive TRSM implementations. Here FLAME/C is interfaced with the DGEMM provided by ITXGEMM.

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

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

Size of the matrix and architecture specifications. Consider the situation where we wish to solve $LX = B$ in the case where together L and B do not fit in the main memory of a processor and are thus stored on disk (out-of-core). Assume also that L fills most of the memory of that processor. Then an out-of-core implementation of TRSM can be achieved by loading most of the memory with L. Next, an algorithm that partitions B by columns, as in Exercise 4.3, can be used to bring blocks of columns of B, (B_1 in the algorithm), into memory, after which some other variant can be used to compute $B_1 := L^{-1}B_1$. Once updated, B_1 is returned to disk.

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

A.4 Stability

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

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

ACKNOWLEDGMENTS

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

REFERENCES

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

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

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

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

DIJKSTRA, E. W. 1976. *A Discipline of Programming*. Prentice-Hall.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

MISRA, J. 1976. Some aspects of the verification of loop computations. *IEEE Trans. Soft. Eng. SE-4*, 6 (Nov.).

MOLER, C., LITTLE, J., AND BANGERT, S. 1987. *Pro-Matlab, User's Guide*. The Mathworks, Inc.

QUINTANA, E. S., QUINTANA, G., SUN, X., AND VAN DE GEIJN, R. 2001. A note on parallel matrix inversion. *SIAM J. Sci. Comput. 22*, 5, 1762–1771.

QUINTANA-ORTÍ, E. S. AND VAN DE GEIJN, R. A. 2003. Formal derivation of algorithms for the triangular Sylvester equation. *ACM Trans. Math. Soft. 29*, 2 (July), 218–243.

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

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

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

Received October 2002; revised June 2003, July 2003, July 2004; accepted August 2004

Parallel Out-of-Core Computation and Updating of the QR Factorization

BRIAN C. GUNTER and ROBERT A. VAN DE GEIJN
The University of Texas at Austin

This article discusses the high-performance parallel implementation of the computation and updating of QR factorizations of dense matrices, including problems large enough to require out-of-core computation, where the matrix is stored on disk. The algorithms presented here are scalable both in problem size and as the number of processors increases. Implementation using the Parallel Linear Algebra Package (PLAPACK) and the Parallel Out-of-Core Linear Algebra Package (POOCLAPACK) is discussed. The methods are shown to attain excellent performance, in some cases attaining roughly 80% of the "realizable" peak of the architectures on which the experiments were performed.

Categories and Subject Descriptors: G.1.10 [**Numerical Analysis**]: Applications; G.4 [**Mathematical Software**]—*Parallel and vector implementations*; J.2 [**Physical Sciences and Engineering**]—*Earth and atmospheric sciences*

General Terms: Algorithms, Theory, Performance

Additional Key Words and Phrases: Linear algebra, dense systems, linear least squares

1. INTRODUCTION

With the recent improvements in memory access, storage capacity, and processing power of high-performance computers, operating on large dense linear systems is not as daunting a task as it has been in the past. One such realm where this new capability has been of extreme value is in the earth sciences, particularly with regards to the determination of high resolution

The authors would like to acknowledge the Texas Advanced Computing Center (TACC) for providing access to the IBM P690s and Cray T3E machines, along with other computing resources, used in the development of this study. Support for this research was provided by NASA grant NAG5-97213 and NSF grant ACI-0203685. Additional support for this work was provided by the NASA Earth System Science Fellowship and the Delores Liebmann Fellowship program.
For more information on PLAPACK and POOCLAPACK visit http://www.cs.texas.edu/users/plapack.
Authors' addresses: B. C. Gunter, Center for Space Research, The University of Texas at Austin, Austin, TX 78712; email: gunter@csr.utexas.edu; R. A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: rvdg@cs.utexas.edu.

gravity field models. The estimation of these models involves the solution of large overdetermined linear least-squares problems, which often contain tens of thousands of parameters and millions of observations [Gunter et al. 2001b; Condi et al. 2003]. Since these systems have the potential to be mildly ill-conditioned, the method chosen to compute the least-squares solution is one utilizing the QR factorization, since it provides greater accuracy than the method of normal equations. To tackle a problem of this size requires the use of an efficient and scalable implementation of the QR factorization that can take advantage of the power of modern day supercomputers. Initially, an in-core parallel implementation was developed [Gunter 2000], but later an out-of-core (OOC) implementation was created to handle even larger problems.

In reporting the results of our research, this article makes the following contributions:

—It reviews the standard techniques for the high-performance implementation of the QR factorization based on the Householder transformation [Dongarra et al. 1986; Bischof and Van Loan 1987; Schreiber and Van Loan 1989; Elmroth and Gustavson 1998, 2000, 2001; Bjorck 1996].

—It extends these techniques to the problem of updating the QR factorization as additional batches of observations are added to the system [Dongarra et al. 1979].

—It demonstrates how the techniques used to update a QR factorization can be used to implement an OOC QR factorization algorithm that is more scalable than other previously proposed OOC approaches for this problem [D'Azevedo and Dongarra 1997; Rabani and Toledo 2001; Toledo and Rabani 2002]. In particular, while it was previously observed that so-called tiled approaches are more scalable than so-called "slab" approaches for the OOC computation of the Cholesky factorization [Toledo and Gustavson 1996; Reiley and van de Geijn 1999; Reiley 1999; Gunter et al. 2001a] and a possibly unstable variant of the LU factorization [Scott 1993], only nonscalable slab approaches had been previously proposed for the OOC QR factorization.

—It discusses a parallel implementation of the algorithms using the Parallel Linear Algebra Package (PLAPACK) [van de Geijn 1997] and its out-of-core extension, the Parallel Out-of-Core Linear Algebra Package (POOCLAPACK) [Reiley and van de Geijn 1999; Reiley 1999; Gunter et al. 2001a; Alpatov et al. 1997].

—It reports excellent performance attained on massively parallel distributed memory supercomputers.

While we reported initial performance results for the implementations in a previous (conference) paper [Gunter et al. 2001a], this article goes into considerably more depth.

This article is structured as follows: Section 2 briefly describes the notation used. Section 3 reviews the QR factorization using Householder transformations. This includes a discussion of the block algorithm as well as the QR factorization's application to the least squares problem. Section 4 discusses how

the standard QR factorization can be updated when new information becomes available. This technique later becomes a key component of the OOC algorithm, which is given in Section 5. The actual parallel implementation is discussed in Section 6. Results from experiments are reported in Section 7. Section 8 provides some final thoughts and conclusions.

2. NOTATION

In this article we adopt the following conventions: Matrices, vectors, and scalars are denoted by upper-case, lower-case, and lower-case Greek letters, respectively. The identity matrix will be denoted by I and e_1 will denote the first column of the identity matrix (in other words, the vector with first element equal to unity and all other elements equal to zero). The dimensions and lengths of such matrices and vectors will generally be obvious from the context.

Algorithms in this article are given in a notation that we have recently adopted as part of the FLAME project [Gunnels et al. 2001; Quintana-Ortí and van de Geijn 2003]. The double lines in the partitioned matrices and vectors relate to how far into the matrices and vectors the computation has proceeded, indicating which parts are in their factored form and which parts are in their original form. We believe the notation to be intuitive, but suggest that the reader consult some of these related papers for further clarification.

3. COMPUTING THE QR FACTORIZATION VIA HOUSEHOLDER TRANSFORMATIONS

Given an $m \times n$ real-valued matrix A, with $m \geq n$, the QR factorization is given by

$$A = QR,$$

where the $m \times m$ matrix Q has mutually orthonormal columns ($Q^T Q = I$) and the $m \times n$ matrix R is upper triangular.

There are many different methods for computing the QR factorization, including those based on Givens rotations, orthogonalization via Gram-Schmidt and Modified Gram-Schmidt, and Householder transformations [Golub and Van Loan 1996; Watkins 1991]. For dense matrices, the method of choice depends largely on how the factorization is subsequently used, the stability of the system, and the dimensions of the matrix. For problems where $m \gg n$, the method based on Householder transformations is typically the algorithm of choice, especially when Q does not need to be explicitly computed.

3.1 Householder Transformations (Reflectors)

Given the real-valued vector x of length m, partition

$$x = \left(\frac{\chi_1}{x_2} \right),$$

where χ_1 equals the first element of x.

$$\textbf{Partition}\quad A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \text{ and } b \rightarrow \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right)$$
$$\textbf{where}\quad A_{TL} \text{ is } 0 \times 0 \text{ and } b_T \text{ has } 0 \text{ elements}$$

$$\textbf{while}\ \ n(A_{BR}) \neq 0\ \ \textbf{do}$$
$$\textbf{Repartition}$$
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & a_{01} & A_{02} \\ \hline\hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline\hline A_{20} & a_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline\hline \beta_1 \\ \hline\hline b_2 \end{array}\right)$$
$$\textbf{where}\quad \alpha_{11} \text{ and } \beta_1 \text{ are scalars}$$

$$\left[\left(\begin{array}{c} 1 \\ \hline u_2 \end{array}\right), \eta, \beta_1\right] := h\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$
$$\alpha_{11} := \eta$$
$$a_{21} := u_2$$
$$w^T := a_{12}^T + u_2^T A_{22}$$
$$\left(\begin{array}{c} a_{12}^T \\ \hline\hline A_{22} \end{array}\right) := \left(\begin{array}{c} a_{12}^T - \beta_1 w^T \\ \hline\hline A_{22} - \beta_1 u_2 w^T \end{array}\right)$$

$$\textbf{Continue with}$$
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline\hline A_{20} & a_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline\hline b_2 \end{array}\right)$$
$$\textbf{enddo}$$

Fig. 1. Unblocked Householder QR factorization.

We define the Householder vector associated with x as the vector

$$u = \left(\begin{array}{c} 1 \\ \hline x_2/\nu_1 \end{array}\right),$$

where $\nu_1 = \chi_1 + \text{sign}(\chi_1)\|x\|_2$. If $\beta = \frac{2}{u^T u}$ then $(I - \beta u u^T)x = \eta e_1$, annihilating all but the first element of x. Here $\eta = -\text{sign}(\chi_1)\|x\|_2$. The transformation $I - \beta u u^T$, with $\beta = 2/u^T u$ is referred to as a Householder transformation or reflector.

Let us introduce the notation $[u, \eta, \beta] := h(x)$ as the computation of the above mentioned η, u, and β from vector x and the notation $H(x)$ for the transformation $(I - \beta u u^T)$ where $[u, \eta, \beta] = h(x)$. An important feature of $H(x)$ is that it is orthonormal $(H(x)^T H(x) = H(x)H(x)^T = I)$ and symmetric $(H(x)^T = H(x))$.

3.2 A Simple Algorithm for the QR Factorization via Householder Transformations

The computation of the QR factorization commences as described in Figure 1. The idea is that Householder transformations are computed to successively annihilate elements below the diagonal of matrix A one column at a time. The Householder vectors are stored below the diagonal over the elements of A that have been so annihilated. Upon completion, matrix R has overwritten the upper triangular part of the matrix, while the Householder vectors are stored in the lower trapezoidal part of the matrix. The scalars β discussed above are stored in the vector b.

$$\textbf{Partition}\ \ A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \text{ and } T \rightarrow \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right)$$
$$\textbf{where}\quad A_{TL} \text{ is } 0 \times 0 \text{ and } T_T \text{ has 0 rows}$$

$$\textbf{while}\ \ n(A_{BR}) \neq 0\ \ \textbf{do}$$
$$\textbf{Determine block size } k$$
$$\textbf{Repartition}$$
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right) \rightarrow \left(\begin{array}{c} T_0 \\ \hline\hline T_1 \\ \hline\hline T_2 \end{array}\right)$$
$$\textbf{where}\quad A_{11} \text{ is } k \times k \text{ and } T_1 \text{ has } k \text{ rows}$$

$$\left[\left(\frac{A_{11}}{A_{21}}\right), \eta, b_1\right] := \left[\left(\frac{\{Y\backslash R\}_{11}}{Y_{21}}\right), \eta, b_1\right] = QR\left(\left(\frac{A_{11}}{A_{21}}\right)\right)$$
$$\text{Compute } T_1 \text{ from } \left[\left(\frac{\{Y\backslash R\}_{11}}{Y_{21}}\right), \eta, b_1\right]$$
$$\left(\frac{A_{12}}{A_{22}}\right) := \left(I + \left(\frac{Y_{11}}{Y_{21}}\right) T_1^T \left(\begin{array}{c|c} Y_{11}^T & Y_{21}^T \end{array}\right)\right) \left(\frac{A_{12}}{A_{22}}\right)$$

$$\textbf{Continue with}$$
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right) \leftarrow \left(\begin{array}{c} T_0 \\ \hline T_1 \\ \hline\hline T_2 \end{array}\right)$$
$$\textbf{enddo}$$

Fig. 2. Blocked Householder QR factorization.

If the matrix Q is explicitly desired, it can be formed by computing the first n columns of $H_1 H_2 \cdots H_n$ where H_i equals the ith Householder transformation computed as part of the factorization described above. In our application, we do not need to form Q explicitly and thus will not discuss the issue further.

3.3 A High-Performance (Blocked) Algorithm for the QR Factorization

It is well-known that high performance can be achieved in a portable fashion by casting algorithms in terms of matrix-matrix multiplication [Anderson et al. 1992; Dongarra et al. 1990, 1991]. We now review how to do so for the QR factorization [Schreiber and Van Loan 1989; Anderson et al. 1992].

Two observations play a key role:

—Let $u_1, \ldots, u_k$ equal the first k Householder vectors computed as part of the factorization, and $\beta_1, \ldots, \beta_k$ the corresponding scalars. Then $H_1 H_2 \cdots H_k = I + Y T Y^T$ where Y is a $n \times k$ unit lower-trapezoidal matrix, T is a $k \times k$ upper-triangular matrix, and the jth column of the lower-trapezoidal part of Y equals u_j.

—The QR factorization of the first k columns of A yields the same k vectors u_k and the same values in the upper triangular part of those k columns as would a full QR factorization.

Notice that this suggests the blocked algorithm given in Figure 2.

NOTE 1. *Notice that the algorithm stores the "T" matrices that are part of the block Householder transformation $I + YTY^T$. This avoids having to recompute those matrices as part of the OOC implementation of the QR factorization, and results in a small but noticeable increase in performance [Elmroth and Gustavson 1998, 2000, 20001]. It should be noted that when factoring tall, thin matrices, the performance gain from saving the T matrices is much greater, in some cases a factor of three or more. The cases examined in this study, in particular the OOC algorithm presented later, work primarily on roughly square matrices, which explains why the performance gain is not as dramatic. In addition, while not implemented for this study, further optimizations can be gained for certain types of problems by formulating the T matrices in terms of Level-3 operations [Elmroth and Gustavson 2000] as opposed to the traditional method, which only incorporates Level-2 operations [Schreiber and Van Loan 1989].*

3.4 Solving Multiple Linear Least-Squares Problems

Given a real-valued $m \times n$ matrix A and vector y of length m, the linear least-squares problem is generally stated as

$$\min_x \| y - Ax \|_2,$$

where the desired result is a vector x that minimizes the above expression. It is well-known that the minimizing vector x can be found by computing the QR factorization $A = QR$, computing $z = Q^T y$, and solving $Rx = z_T$ where z_T denotes the first n elements of z.

Alternatively, one can think of this as follows: Append y to A to form $(A \mid y)$. Compute the QR factorization $A = QR$, storing the Householder vectors and R over A. Update y by applying the Householder transformations used to compute R to vector y, which overwrites y with z. Finally, solve $Rx = z_T$ with the first n elements of the updated y. This second approach is reminiscent of how a linear system can be solved by appending the right-hand-side vector to the system and performing an LU factorization (or, equivalently, Gaussian elimination) on the appended system, followed by a back-substitution step.

Finally, if there exists a set of right-hand-sides, one can simultaneously solve a linear least-squares problem with A and columns of B by the following approach: Append B to A to form $(A \mid B)$. Compute the QR factorization $A = QR$, storing the Householder vectors and R over A. Update B by applying the Householder transformations used to compute R to matrix B, which overwrites B with Z. Finally, solve $RX = Z_T$ with the first n rows of the updated B. It is this second operation with a right-hand-side B that we will encounter in the OOC implementation of the QR factorization. An algorithm for the first, forward substitution-like, step is given in Figure 3.

NOTE 2. *Again, because we store the "T" matrices that are part of the block Householder transformation $I + YTY^T$, they need not be recomputed as part of the "forward substitution" step on matrix B (see Note 1 for details).*

$$\textbf{Partition } A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right), B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \text{ and } T \rightarrow \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right)$$
$$\textbf{where } A_{TL} \text{ is } 0 \times 0 \text{ and } B_T \text{ and } T_T \text{ have 0 rows}$$

while $n(A_{BR}) \neq 0$ **do**
Determine block size k
Repartition

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right),$$
$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline\hline B_2 \end{array}\right), \text{ and } \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right) \rightarrow \left(\begin{array}{c} T_0 \\ \hline\hline T_1 \\ \hline\hline T_2 \end{array}\right)$$
$$\textbf{where } A_{11} \text{ is } k \times k \text{ and } B_1 \text{ and } T_1 \text{ have } k \text{ rows}$$

$$\left(\begin{array}{c} B_1 \\ \hline\hline B_2 \end{array}\right) := \left(I + \left(\begin{array}{c} Y_{11} \\ \hline\hline Y_{21} \end{array}\right) T_1^T \left(\begin{array}{c|c} Y_{11}^T & Y_{21}^T \end{array}\right)\right) \left(\begin{array}{c} B_1 \\ \hline\hline B_2 \end{array}\right)$$

NOTE: Here Y_{11} refers to the lower triangular part of A_{11} and Y_{21} to A_{21}

Continue with

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right),$$
$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline\hline B_2 \end{array}\right), \text{ and } \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right) \leftarrow \left(\begin{array}{c} T_0 \\ \hline\hline T_1 \\ \hline\hline T_2 \end{array}\right)$$

enddo

Fig. 3. Blocked update of the right-hand-side matrix B using a "forward substitution-like" approach.

4. UPDATING THE QR FACTORIZATION

Frequently, the linear equations used in the least squares problem are collected incrementally. For example, if the observations from a particular instrument are only collected or contributed on a monthly basis, it would be useful to combine each new batch of data into the existing solution without having to recombine all of the previous data. We now review how the QR factorization can be updated as additional batches of equations (i.e., observations) become available [Dongarra et al. 1979; Golub and Van Loan 1996; Watkins 1991].

4.1 Factorization

Let us assume that we have computed Q and R such that $A = QR$, overwriting A with the Householder vectors and upper triangular matrix R, and storing the "T" matrices in matrix T. Thus, we have available $A = \{Y \backslash R\}$ and T. Now, consider the QR factorization of matrix

$$\left(\begin{array}{c} A \\ \hline\hline C \end{array}\right) = \bar{Q}\bar{R}. \tag{1}$$

$$\textbf{Partition}\quad R \rightarrow \left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array}\right),\ C \rightarrow \left(\begin{array}{c||c} C_L & C_R \end{array}\right),\ \text{and}\ b \rightarrow \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right)$$

where R_{TL} and C_L are 0×0 and b_T has 0 elements

while $n(R_{BR}) \neq 0$ **do**

Repartition

$$\left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} R_{00} & r_{01} & R_{02} \\ \hline\hline r_{10}^T & \rho_{11} & r_{12}^T \\ \hline\hline R_{20} & r_{21} & R_{22} \end{array}\right),$$

$$\left(\begin{array}{c||c} C_L & C_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_0 & c_1 & C_2 \end{array}\right),\ \text{and}\ \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline\hline \beta_1 \\ \hline\hline b_2 \end{array}\right)$$

where ρ_{11} and β_1 are scalars and c_1 is a column

$$\left[\left(\begin{array}{c} 1 \\ \hline\hline u_2 \end{array}\right), \eta, \beta_1\right] := h\left(\begin{array}{c} \rho_{11} \\ \hline\hline c_1 \end{array}\right)$$

$$\rho_{11} := \eta$$

$$c_1 := u_2$$

$$w^T := r_{12}^T + u_2^T C_2$$

$$\left(\begin{array}{c} r_{12}^T \\ \hline\hline C_2 \end{array}\right) := \left(\begin{array}{c} r_{12}^T - \beta_1 w^T \\ \hline\hline C_2 - \beta_1 u_2 w^T \end{array}\right)$$

Continue with

$$\left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c||c|c} R_{00} & r_{01} & R_{02} \\ \hline\hline r_{10}^T & \rho_{11} & r_{12}^T \\ \hline\hline R_{20} & r_{21} & R_{22} \end{array}\right),$$

$$\left(\begin{array}{c||c} C_L & C_R \end{array}\right) \leftarrow \left(\begin{array}{c||c|c} C_0 & c_1 & C_2 \end{array}\right),\ \text{and}\ \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline\hline \beta_1 \\ \hline\hline b_2 \end{array}\right)$$

enddo

Fig. 4. Unblocked update to a QR factorization.

A key observation is that the QR factorization of

$$\left(\begin{array}{c} R \\ \hline\hline C \end{array}\right) \tag{2}$$

produces the same upper triangular matrix $\bar{R}$ as does the factorization in (1). If we are not interested in explicitly forming $\bar{Q}$ and are satisfied with storing the Householder vectors required to first compute the QR factorization of A and next the QR factorization in (2), then we have an approach for computing the QR factorization of an updated system. The unblocked and blocked algorithm for doing so is given in Figures 4 and 5, respectively.

NOTE 3. *Notice that the algorithm is explicitly designed to take advantage of the zeros below the diagonal of R. As a result, factoring A followed by an update of the factorization requires essentially no more computation than the factorization in (1). Also, the Householder vectors that are stored below the diagonal are not overwritten. It is the case that an additional vector b is required to store the "β"s for the unblocked algorithm and an additional matrix is required to store the triangular "T" matrices for the blocked algorithm.*

$$\textbf{Partition}\quad R \rightarrow \left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array}\right),\ C \rightarrow \left(\begin{array}{c||c} C_L & C_R \end{array}\right),\ \text{and}\ T \rightarrow \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right)$$

$$\textbf{where}\quad R_{TL}\ \text{and}\ C_L\ \text{are}\ 0 \times 0\ \text{and}\ T_T\ \text{has 0 rows}$$

$$\textbf{while}\ \ n(R_{BR}) \neq 0\ \ \textbf{do}$$

$$\textbf{Determine block size}\ k$$

$$\textbf{Repartition}$$

$$\left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} R_{00} & R_{01} & R_{02} \\ \hline\hline R_{10} & R_{11} & R_{12} \\ \hline\hline R_{20} & R_{21} & R_{22} \end{array}\right),$$

$$\left(\begin{array}{c} C_L \\ \hline\hline C_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_0 & C_1 & C_2 \end{array}\right),\ \text{and}\ \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right) \rightarrow \left(\begin{array}{c} T_0 \\ \hline\hline T_1 \\ \hline\hline T_2 \end{array}\right)$$

$$\textbf{where}\quad A_{11}\ \text{is}\ k \times k,\ C_1\ \text{has}\ k\ \text{columns and}\ T_1\ \text{has}\ k\ \text{rows}$$

$$\left[\left(\begin{array}{c} R_{11} \\ \hline\hline C_1 \end{array}\right), \eta, b_1\right] := \left[\left(\begin{array}{c} \{0\backslash R\}_{11} \\ \hline\hline Y_1 \end{array}\right), \eta, b_1\right] = QR\left(\left(\begin{array}{c} R_{11} \\ \hline\hline C_1 \end{array}\right)\right)$$

$$\text{Compute}\ T_1\ \text{from}\ \left[\left(\begin{array}{c} I \\ \hline\hline Y_1 \end{array}\right), \eta, b_1\right]$$

$$\left(\begin{array}{c} R_{12} \\ \hline\hline C_2 \end{array}\right) := \left(I + \left(\begin{array}{c} I \\ \hline\hline Y_1 \end{array}\right) T_1^T \left(\begin{array}{c|c} I & Y_1^T \end{array}\right)\right) \left(\begin{array}{c} R_{12} \\ \hline\hline C_2 \end{array}\right)$$

$$\textbf{Continue with}$$

$$\left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} R_{00} & R_{01} & R_{02} \\ \hline\hline R_{10} & R_{11} & R_{12} \\ \hline\hline R_{20} & R_{21} & R_{22} \end{array}\right),$$

$$\left(\begin{array}{c||c} C_L & C_R \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} C_0 & C_1 & C_2 \end{array}\right),\ \text{and}\ \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right) \leftarrow \left(\begin{array}{c} T_0 \\ \hline\hline T_1 \\ \hline\hline T_2 \end{array}\right)$$

$$\textbf{enddo}$$

Fig. 5. Blocked update to a QR factorization.

4.2 Solving Multiple Linear Least-Squares Problems

If we now wish to compute multiple linear least-squares solutions, one for each of the systems of linear equations defined by picking one column of the right-hand-side of

$$\left(\begin{array}{c} A \\ \hline\hline C \end{array}\right) X = \left(\begin{array}{c} B \\ \hline\hline D \end{array}\right),$$

the following approach will yield the desired result:

—Append $\left(\begin{array}{c|c} A & B \\ \hline\hline C & D \end{array}\right)$.

—Overwrite A with its factors $\{Y \backslash R\}$, also computing matrix T, as in Figure 2.

—Overwrite $\left(\begin{array}{c} R \\ \hline\hline C \end{array}\right)$ with $\left(\begin{array}{c} \bar{R} \\ \hline\hline Y^{(C)} \end{array}\right)$, also computing $T^{(C)}$ as in Figure 5.

—Update B by forward substitution as in Figure 3.

—Update $\left(\begin{array}{c} B \\ \hline\hline D \end{array}\right)$ by forward substitution using the Householder transformations computed as part of the update of R, as in Figure 6.

—Solve $\bar{R}X = B_T$, where B_T denotes the top n rows of the updated matrix B.

Partition $B \rightarrow \left(\frac{B_T}{B_B}\right)$, $C \rightarrow \left(\begin{array}{c|c} C_L & C_R \end{array}\right)$, and $T \rightarrow \left(\frac{T_T}{T_B}\right)$
where C_L has 0 columns, and B_T and T_T have 0 rows

while $n(C_R) \neq 0$ **do**
Determine block size k
Repartition

$$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right),$$
$$\left(\frac{B_T}{B_B}\right) \rightarrow \left(\frac{B_0}{\frac{B_1}{B_2}}\right), \text{ and } \left(\frac{T_T}{T_B}\right) \rightarrow \left(\frac{T_0}{\frac{T_1}{T_2}}\right)$$

where C_1 has k columns and B_1 and T_1 have k rows

$$\left(\frac{B_1}{D}\right) := \left(I + \left(\frac{I}{C_1}\right) T_1^T \left(\begin{array}{c|c} I & C_1^T \end{array}\right)\right) \left(\frac{B_1}{D}\right)$$

Continue with

$$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right),$$
$$\left(\frac{B_T}{B_B}\right) \leftarrow \left(\frac{B_0}{\frac{B_1}{B_2}}\right), \text{ and } \left(\frac{T_T}{T_B}\right) \leftarrow \left(\frac{T_0}{\frac{T_1}{T_2}}\right)$$

enddo

Fig. 6. Forward substitution consistent with the QR factorization of an updated matrix.

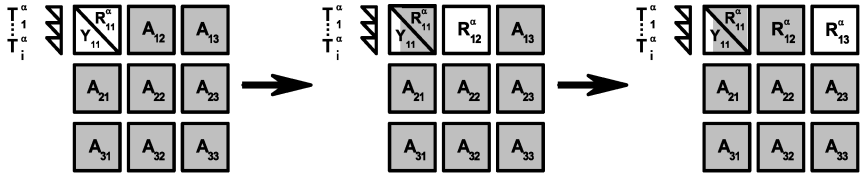
5. OUT-OF-CORE ALGORITHMS

Having now described the in-core algorithm, we can apply a similar strategy for problems that are too large to fit in the available memory of the machine. To deal with these problems, we have developed an out-of-core algorithm that allows us to store the bulk of the matrix components on disk, while only working on select pieces in-core at any one time. The algorithm we will outline here is unique in that it is both scalable and efficient.

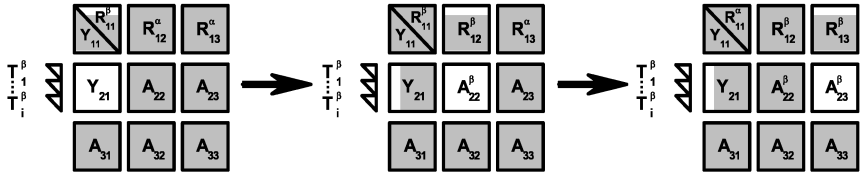
5.1 Out-of-Core QR Factorization

Traditional OOC algorithms of the QR factorization have used a slab approach, in which the OOC matrix is processed by bringing into memory one or more slabs (blocks of columns) of the matrix at a time [Coleman et al. 1992; Klimkowski and van de Geijn 1995; D'Azevedo and Dongarra 1997; Toledo and Gustavson 1996; Scott 1993; Toledo and Rabani 2002]. The problem with this technique is that it is inherently not scalable in the following sense: As the row dimension, m, of A becomes larger and larger, the width of the slab you can bring into memory becomes proportionally smaller. As m reaches into the millions, the number of columns able to be brought into memory numbers only in the dozens, even on today's powerful machines with large memories.

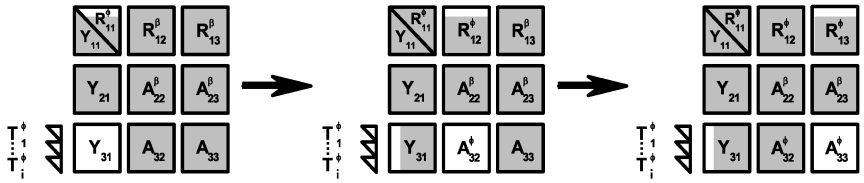
The alternative that we have found to the slab approach is to work with the matrix as a collection of tiles, where a tile is a submatrix that is roughly square. As was shown for the OOC Cholesky factorization [Toledo and Gustavson 1996;



(a) A_{11} is factored using the in-core algorithm, then the remaining tiles in the first row are updated using the Householder vectors stored in the lower triangular portion of A_{11}. The Householder vectors are read and applied in narrow panels of width r.



(b) The second row of tiles is now processed. A_{21} is factored using the modified in-core algorithm. As in step a, the Householder vectors are applied in panels of width r, while the updates to the first row of tiles are done in horizontal panels of height r. Note the need to create a new set of T matrices.



(c) The last tile row is factored, with the first row of tiles receiving its final update. As before, new T matrices are created. The second tile row is not affected.

Fig. 7. Factoring the first row of tiles using the out-of-core approach. Grey regions indicate components that reside on disk.

Reiley and van de Geijn 1999; Reiley 1999; Gunter et al. 2001a], a tiled approach provides true scalability. We will see that the processing of these tiles becomes a simple application of the algorithms in Figures 2, 3, 5, and 6. The high performance achieved with these in-core procedures is maintained when processing the tiles, providing the same benefits to the OOC approach. As the problem size increases, additional tiles are simply added to the system, without adversely affecting performance.

To demonstrate this, we begin in much the same way as we did with the in-core algorithm, except now the matrix A resides entirely on disk and not in memory. We partition the matrix into a series of tiles, as illustrated in Figure 7. For the purposes of this example, we will assume that the matrix A is square and divided into nine tiles of size $t \times t$, forming a 3×3 grid of tiles.

(1) The first tile, A_{11}, is read into memory and factored using the in-core algorithm in Figure 2. Upon completion, A_{11} is written to disk. For now, the "T" matrices created during this step are kept in-core.
Notice that as the dimension t becomes larger, the cost of reading and writing A_{11}($O(t^2)$) improves because it is amortized over the useful computation

($O(t^3)$)—the QR factorization. Consequently, the larger t becomes, the less significant the I/O overhead.

(2) Next, tile A_{12} is brought into memory. It is updated consistent with the factorization of A_{11}, using the Householder vectors that have overwritten the lower triangular part of A_{11} and the "T" matrices still in memory. In other words, the algorithm in Figure 3 is employed. Once updated, A_{12} is written back to disk.
On the surface, it would thus appear that two tiles must be in memory, consequently limiting the tile dimension t. However, a closer look at the update in the body of the loop of the algorithm in Figure 3 shows that only a panel of columns of A_{11}, which can be discarded as soon as it has been used to update A_{12}, needs to be brought into memory. Thus, at most $t \times k$ elements of A_{11} need to be in memory at a time. The cost of bringing these elements into memory is amortized over $O(kt^2)$ computations. Similarly, the $O(t^2)$ cost of bringing A_{12} into memory is amortized over $O(t^3)$ computations. The larger t becomes, the less significant the I/O overhead.
The remaining tiles in the first row are processed similarly. Once the entire first row has been processed, the "T" matrices computed in the factorization of A_{11} can be written to disk.

(3) After processing the first row, A_{21} is brought into memory. It must be updated together with A_{11} according to the algorithm in Figure 5, generating a new set of "T" matrices. Once updated, A_{21} is written to disk, while the newly generated "T" matrices are kept in memory.
Again, it would appear that two tiles must be in memory, thus limiting the tile dimension t. However, the update in the body of the loop of the algorithm in Figure 5 shows that only a panel of rows of A_{11}, which can be written back to disk as soon as it has been used to update A_{21}, needs to be brought into memory. Thus, at most $k \times t$ elements of A_{11} need to be in memory at a time. The cost of bringing these elements into memory is amortized over $O(kt^2)$ computations. Similarly, the $O(t^2)$ cost of bringing A_{21} into memory is amortized over $O(t^3)$ computations. The larger is t, the less significant the I/O overhead.

(4) Once A_{21} is updated, A_{22} is brought into memory, to be updated according to Step 3 above, using the algorithm in Figure 6.
It would appear that now A_{21}, A_{12}, and A_{22} must all be in memory simultaneously. However, the update in the body of the loop in Figure 6 requires only a panel of columns of A_{21} and a panel of rows of A_{12} to be in memory. Thus only A_{22} needs to be kept in memory, while panels of the other two matrices are streamed from disk. The cost of the I/O involved is $O(kt)$ per iteration of the loop, which is amortized over $O(kt^2)$ computations. Meanwhile, the $O(t^2)$ cost of bringing A_{22} into memory is amortized over $O(t^3)$ computations. The larger t becomes, the less significant the I/O overhead.
The remaining tiles in the second row are processed similarly.

(5) The third row of tiles is handled in the same manner as described in Steps 3 and 4. A_{31} is first factored with A_{11}, creating a new set of "T" matrices and overwriting A_{31} with the corresponding Householder vectors. A_{32} is brought

into memory and updated with column panels from A_{31}, while A_{12} in row panels of height k is also updated. The same is done for A_{33} and A_{13}. Note that only the first and third row of tiles are affected by these operations.

(6) Now, Steps 5.1–5 start to repeat: A_{22} is factored as A_{11} was in Step 5.1. The remaining tiles in the second row are processed as described in Step 2. The tiles in the third row below the tiles on the diagonal are processed as in Step 3, and the remaining tiles in the third row are processed as in Steps 4 and 5. After this, it is back to Step 5.1 with A_{33} and so forth.

This process is illustrated in Figure 7. In that figure, it is the unshaded part of the matrix that is in memory at a typical stage of the algorithm.

NOTE 4. *In principle, most of the memory can be dedicated to storing a single $t \times t$ tile. This allows t to be as large as possible, which then improves the indicated ratios of I/O to useful computation.*

5.2 Out-of-Core Updating

To update an existing OOC solution with a new set of equations is straightforward, as the OOC algorithm was designed to handle problems of arbitrary length. The new data is simply divided into the appropriate tiles and combined with the existing solution in the same manner that the 2nd and 3rd rows of tiles were handled in the previous section.

6. IMPLEMENTATION

In this section, we discuss some practical considerations related to the actual implementation of the OOC algorithms.

6.1 Parallel Implementation

So far in this article, parallel implementation has not been explicitly discussed. We now give a few details.

It is well-known that a scalable implementation of dense linear algebra operations requires the use of a so-called two-dimensional matrix distribution [Hendrickson and Womble 1994; Stewart 1990]. Moreover, to ensure load-balance as the active part of the matrix shrinks, an overdecomposition and wrapping of the matrix is typically employed [Lichtenstein and Johnsson 1992; Strazdins 1998; van de Geijn 1997; Dongarra et al. 1994].

The observation now is that if one has parallel implementations of the algorithms in Figures 1–6, then the parallel implementation of the OOC algorithm becomes straightforward. The parallel implementation of the QR factorization is well-understood, and is available as part of our PLAPACK package, as well as part of the Scalable Linear Algebra Package (ScaLAPACK) [Choi et al. 1992]. Since the remaining algorithms are, in essense, merely variations of the QR factorization, we implemented them as modifications of the PLAPACK QR factorization. Further modifications had to be made so that the panels being streamed from disk were read into memory and/or written out to disk at the appropriate time. Our POOCLAPACK package facilitates this read operation. Finally, a routine that manages the processing of the tiles was also written.

6.2 Optimizing I/O Performance

The OOC method described in Section 5 advocates a single-tile method, in which most of memory is dedicated to a single tile. As argued, this is desirable because the I/O overhead decreases as the tile size increases. While this is easy to justify theoretically, in this section we point out a practical consideration that suggests that keeping two tiles in memory may lead to better performance. The two tile approach also leads to a simpler implementation.

First, a few details about the storage of matrices. The matrices assigned locally to each processor as parts of tiles are stored in memory in column-major order. Similarly, on disk, they are stored in column-major order. More precisely, the columns are stored so that if a panel of columns is read by a processor from disk, they are all contiguous in memory. This makes the reading of a tile and of panels of columns relatively cheap, since I/O carries a large start-up cost (latency). In other words, a panel of columns can be read essentially at peak bandwidth. By contrast, the reading of a panel of rows is generally staged as the reading of individual columns of that panel, incurring a latency related cost for each such column. This makes the reading of a panel of rows prohibitively expensive.

The update in Step 2 in Section 5.1 requires only column panels (of Householder vectors) to be read from disk. By contrast, the operations in Steps 3 and 4 require panels of rows to be brought in. Thus, it becomes advantageous to bring the entire tile from which panels of rows are to be used into memory, leading to a two-tile OOC algorithm. It is this approach that was actually implemented by us and used to obtain the performance numbers described in the next section.

NOTE 5. *By transposing tiles above the diagonal after processing, it is possible to implement the one-tile approach while still only reading panels of columns. We did not do so in an effort to keep the implementation simple.*

7. PERFORMANCE

In this section, we demonstrate that the presented algorithm attains very high performance on distributed memory parallel architectures.

7.1 Target Architectures

The POOCLAPACK implementation of the OOC QR factorization and update algorithm is essentially portable to any platform that supports the Message-Passing Interface [Gropp et al. 1994; Snir et al. 1996] and the Basic Linear Algebra Subprograms [Lawson et al. 1979; Dongarra et al. 1988, 1990]. To date, the implementation has been ported to the SGI Origin 3000 and Linux PC cluster environments, in addition to the Cray T3E and IBM P-series systems.

In this article we report performance on two architectures:

—The Cray T3E-600. The system on which the experiments were performed has 272 total processors, each with 128MB of available memory. The T3E operates at a peak theoretical performance of 600 millions of floating point operations per second per processor (MFLOPS/sec/proc). For reference,

the matrix-matrix multiply operation (DGEMM) was benchmarked at 445 MFLOPS/sec/proc for the particular machine used in this study. The BLAS used was provided as part of the Cray Scientific Library. It should also be noted that since the T3E is a true 64-bit platform, all arithmetic was done using 64-bit precision.

—The IBM P690. The system on which the experiments were performed consists of SMP nodes, where each node consists of 16 Power4 (1.3 GHz) processors, with 32 GBytes of available memory. The P690s operate at four FLOPS per cycle for a peak theoretical performance of 5200 MFLOPS/sec/proc, with a DGEMM benchmark of 3723 MFLOPS/sec/proc. IBM's optimized Engineering and Scientific Subroutine Library (ESSL) was used in place of the standard BLAS library. Again, all computation was performed in 64-bit arithmetic.

7.2 Reporting Performance

The operation count for a Householder transform-based QR factorization of an $m \times m$ matrix is given by approximately $\frac{4}{3}m^3$ floating point operations. While our OOC algorithm requires more operations, due to the accumulation and application of the "T" matrices, it is this operation count that represents the *useful* computation.

Thus, given $T_p(m)$, the time in seconds required on p processors to factor an $m \times m$ matrix, the rate in MFLOPS/sec/proc at which the processors compute is given by the formula

$$R_p(m) = \frac{\frac{4}{3}m^3}{T_p(m)} \times \frac{10^{-6}}{p}$$

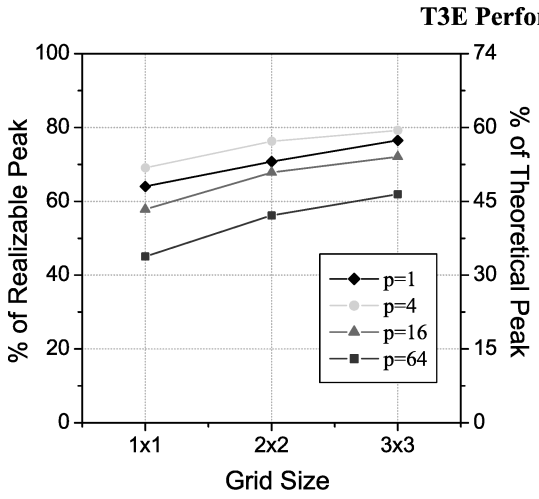
Now, since the bulk of the computation is cast in terms of local matrix-matrix multiplications, the upper bound on $R_p(m)$ is given by the rate in MFLOPS/sec attained by DGEMM, which we will denote by R_{dgemm}. We consider this to be the peak performance that can be attained per processor, or the "realizable" peak of the system. The performance of our OOC implementation will be reported as a percentage of this realizable peak by the ratio

$$\frac{R_p(m)}{R_{dgemm}}.$$

Depending on the architecture, this realizable peak is 70–99% of the theoretical peak of the processor, which is defined by the clock speed multiplied by the number of floating point operations that can be performed per clock cycle. We believe that reporting performance relative to the realizable peak gives a clearer insight into the overhead incurred by the parts of the QR factorization that are not cast in terms of matrix-matrix multiplication, the overhead due to the parallelization, and the overhead due to I/O.

7.3 Results

Figure 8 illustrates the performance of the OOC algorithm. In our experiment, we vary both the number of processors used and the problem size in the following way: Parameter t is chosen as the dimension of the tiles that will

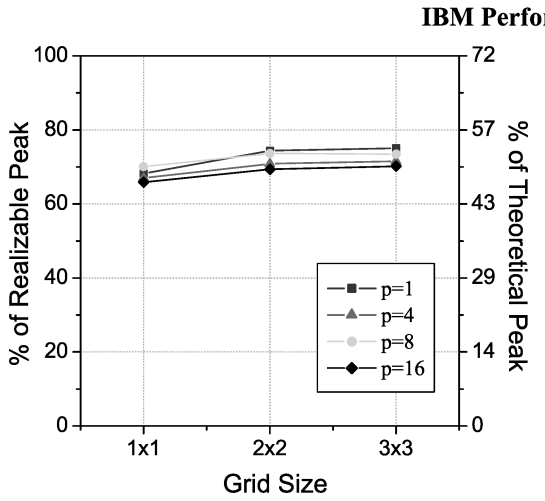


Problem Size

# Procs	1	4	8	16
Tile Size	2088	4704	8448	18432
n (1 x 1)	2088	4704	8448	18432
n (2 x 2)	4176	9408	16896	36864
n (3 x 3)	6264	14112	25344	55296

Compute Time (sec.)

# Procs	1	4	8	16
n (1 x 1)	43	112	195	651
n (2 x 2)	308	818	1332	4176
n (3 x 3)	962	2657	4231	12776



Problem Size

# Procs	1	4	8	16
Tile Size	7900	21000	30000	42000
n (1 x 1)	7900	21000	30000	42000
n (2 x 2)	15800	42000	60000	84000
n (3 x 3)	23700	63000	90000	126000

Compute Time (sec.)

# Procs	1	4	8	16
n (1 x 1)	259	1238	1726	2518
n (2 x 2)	1900	9365	13130	19128
n (3 x 3)	6349	31286	44439	63770

Fig. 8. Performance of the OOC algorithm on a Cray T3E and IBM P690.

be kept in memory. Naturally, as the number of processors increases, the total available memory increases, and t can be increased. Factorizations of problems of size 1×1 tiles through 3×3 tiles were subsequently timed (reported as the Grid Size along the x-axis). The curves connect the data points corresponding to the number of processors indicated in the legend. As the figure shows, the performance is quite respectable.

It is interesting to note that as the problem size becomes larger, the performance improves. This can be explained by the fact that as the problem size increases, more of the computation is in the operations in Figures 3 and 6, which casts more of the computation in matrix-matrix multiplication, the operation that attains the highest performance.

There is a noticable difference between the two architectures regarding scalability as the number of processors is increased. This can largely be attributed

to the fact that the I/O performance of the specific Cray T3E used for these experiments becomes a bottleneck as more processors access the disk simultaneously.

7.4 Further Possible Improvements

The use of asynchronous I/O (i.e., overlapping I/O with computation) was explored in a previous study on parallel OOC implementation of the Cholesky factorization [Reiley and van de Geijn 1999; Reiley 1999; Gunter et al. 2001a]. While it was determined that a slight performance increase could be achieved on machines with slower I/O bandwidths, the complexity of the code required to do this was considered prohibitive for the algorithms presented in this article. Advances in I/O technology with newer high performance machines also render this performance increase practically negligible. Consequently, asynchronous I/O was not used to achieve the performance numbers described in Figure 8.

While the above results were obtained using the Cray T3E and IBM P690, it should be noted that the performance of the algorithm on other platforms is comparable when examining the speed as a percentage of the realizable peak.

8. CONCLUSION

We have demonstrated that a modification of the standard in-core QR factorization algorithm, combined with a tile-based approach for out-of-core implementations, results in a highly efficient and powerful method for computing QR factorizations of large, dense matrices. We believe that the resulting implementation is unique in that it is scalable both as the number of processors is increased and as, for a fixed number of processors, the problem size is increased. The performance of these algorithms is impressive, reaching roughly 80% of the "realizable" peak in some cases.

The application of these algorithms has already proven valuable to the Earth Sciences, particularly with regard to the determination of the Earth's gravity field [Gunter 2000; Gunter et al. 2001b; Condi et al. 2003]. In addition, the tile-based approach is well suited for other types of dense linear algebra operations, such as the Cholesky decomposition [Gunter et al. 2001a; Reiley and van de Geijn 1999; Reiley 1999].

9. ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referees for providing their time and many insightful comments in reviewing this work.

REFERENCES

Alpatov, P., Baker, G., Edwards, C., Gunnels, J., Morrow, G., Overfelt, J., van de Geijn, R., and Wu, Y.-J. J. 1997. PLAPACK: Parallel linear algebra package – design overview. In *Proceedings of SC97*.

Anderson, E., Bai, Z., Demmel, J., Dongarra, J. E., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A. E., Ostrouchov, S., and Sorensen, D. 1992. *LAPACK Users' Guide*. SIAM, Philadelphia.

Bischof, C. and Van Loan, C. 1987. The WY representation for products of householder matrices. *SIAM J. Sci. Stat. Comput. 8*(1):s2-s13.

BJORCK, A. 1996. *Numerical Methods for Least Squares Problems*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA.

CHOI, J., DONGARRA, J. J., POZO, R., AND WALKER, D. W. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society Press, 120–127.

COLEMAN, R., LEBACK, B., NORIN, R., SCOTT, D., AND DE HOUTEN, K. V. 1992. Soz - a dense, out-of-core solver with partial pivoting for the iPSC/860: A case history. In *1992 Annual Users Conference*.

CONDI, F., GUNTER, B., RIES, J., AND TAPLEY, B. 2003. Combining sea surface and terrestrial gravity data for global geopotential modelling and geoid determination. In *Eos Trans. AGU*, *84*(46), Fall Meet. Suppl., Abstract G31A-06.

D'AZEVEDO, E. F. AND DONGARRA, J. J. 1997. The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines. LAPACK Working Note 118 CS-97-247, University of Tennessee, Knoxville. (January)

DONGARRA, J., KAUFMANN, L., AND HAMMARLING, S. 1986. Squeezing the most out of eigenvalue solvers on high-performance computers. *Linear Algebra and It Applications 77*:113–136.

DONGARRA, J., VAN DE GEIJN, R., AND WALKER, D. 1994. Scalability issues affecting the design of a dense linear algebra library. *J. Parallel Distrib. Comput. 22*, 3 (September).

DONGARRA, J. J., BUNCH, J. R., MOLER, C. B., AND STEWART, G. W. 1979. *LINPACK Users' Guide*. SIAM, Philadelphia.

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

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

DONGARRA, J. J., DUFF, I. S., SORENSEN, D. C., AND VAN DER VORST, H. A. 1991. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA.

ELMROTH, E. AND GUSTAVSON, F. G. 1998. New serial and parallel recursive QR factorization algorithms for SMP systems. In *PARA*. 120–128.

ELMROTH, E. AND GUSTAVSON, F. G. 2000. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM J. Res. Dev. 44*, 4 (July), 605–624.

ELMROTH, E. AND GUSTAVSON, F. G. 2001. A faster and simpler recursive algorithm for the LAPACK routine DGELS. *BIT 41*, 5, 936–949.

GOLUB, G. H. AND VAN LOAN, C. F. 1996. *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, Baltimore, MD.

GROPP, W., LUSK, E., AND SKJELLUM, A. 1994. *Using MPI*. The MIT Press.

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

GUNTER, B. C. 2000. Parallel least squares analysis of simulated GRACE data. CSR Technical Memoranda CSR-TM-00-05, The Center for Space Research, The University of Texas at Austin.

GUNTER, B. C., REILEY, W. C., AND VAN DE GEIJN, R. A. 2001a. Parallel out-of-core Cholesky and QR factorizations with POOCLAPACK. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society.

GUNTER, B. C., TAPLEY, B. D., AND VAN DE GEIJN, R. A. 2001b. Advanced parallel least squares algorithms for GRACE data processing. In *Proceedings of the International Association of Geodesy (IAG) Conference*. Budapest, Hungary.

HENDRICKSON, B. A. AND WOMBLE, D. E. 1994. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Stat. Comput. 15*, 5, 1201–1226.

KLIMKOWSKI, K. AND VAN DE GEIJN, R. 1995. Anatomy of an out-of-core dense linear solver. In *Proceedings of the International Conference on Parallel Processing 1995*. Vol. III—Algorithms and Applications. 29–33.

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

LICHTENSTEIN, W. AND JOHNSSON, S. L. 1992. Block-cyclic dense linear algebra. Tech. Rep. TR-04-92, Harvard University, Center for Research in Computing Technology. Jan.

QUINTANA-ORTÍ, E. S. AND VAN DE GEIJN, R. A. 2003. Formal derivation of algorithms for the triangular Sylvester equation. *ACM Trans. Math. Soft. 29*, 2 (July), 218–243.

RABANI, E. AND TOLEDO, S. 2001. Out-of-core SVD and QR decompositions. In *In Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing (PARA)*. Norfolk, Virginia.

REILEY, W. C. 1999. Efficient parallel out-of-core implementation of the Cholesky factorization. Tech. Rep. CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin. (December) Undergraduate Honors Thesis.

REILEY, W. C. AND VAN DE GEIJN, R. A. 1999. POOCLAPACK: Parallel Out-of-Core Linear Algebra Package. Tech. Rep. CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin. (November)

SCHREIBER, R. AND VAN LOAN, C. 1989. A storage-efficient WY representation for products of householder transformations. *SIAM J. Sci. Stat. Comput. 10*, 1 (January), 53–57.

SCOTT, D. S. 1993. Parallel I/O and solving out-of-core systems of linear equations. In *Proceedings of the 1993 DAGS/PC Symposium*. Dartmouth Institute for Advanced Graduate Studies, Hanover, NH, 123–130.

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

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

STRAZDINS, P. 1998. Optimal load balancing techniques for block-cyclic decompositions for matrix factorization. Tech. Rep. TR-CS-98-10, Canberra 0200 ACT, Australia.

TOLEDO, S. AND GUSTAVSON, F. G. 1996. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computation. In *Proceedings of IOPADS '96*.

TOLEDO, S. AND RABANI, E. 2002. Very large electronic structure calculations using an out-of-core filter-diagonalization method. *J. Comp. Phys. 180*, 256–269.

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

WATKINS, D. 1991. *Fundamentals of Matrix Computations*, 2nd ed. J. Wiley and Sons, New York.

Received July 2003; revised January 2004 and June 2004; accepted June 2004

Representing Linear Algebra Algorithms in Code: The FLAME Application Program Interfaces

PAOLO BIENTINESI
The University of Texas at Austin
ENRIQUE S. QUINTANA-ORTí
Universidad Jaume I
and
ROBERT A. VAN DE GEIJN
The University of Texas at Austin

In this article, we present a number of Application Program Interfaces (APIs) for coding linear algebra algorithms. On the surface, these APIs for the MATLAB M-script and C programming languages appear to be simple, almost trivial, extensions of those languages. Yet with them, the task of programming and maintaining families of algorithms for a broad spectrum of linear algebra operations is greatly simplified. In combination with our Formal Linear Algebra Methods Environment (FLAME) approach to deriving such families of algorithms, dozens of algorithms for a single linear algebra operation can be derived, verified to be correct, implemented, and tested, often in a matter of minutes per algorithm. Since the algorithms are expressed in code much like they are explained in a classroom setting, these APIs become not just a tool for implementing libraries, but also a valuable tool for teaching the algorithms that are incorporated in the libraries. In combination with an extension of the Parallel Linear Algebra Package (PLAPACK) API, the approach presents a migratory path from algorithm to MATLAB implementation to high-performance sequential implementation to parallel implementation. Finally, the APIs are being used to create a repository of algorithms and implementations for linear algebra operations, the FLAME Interface REpository (FIRE), which already features hundreds of algorithms for dozens of commonly encountered linear algebra operations.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]—*Algorithm design and analysis*; *efficiency*; *user interfaces*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain*

Support for this research was provided by NSF grant ACI-0305163. Additional support for this work came from the Visiting Researcher program of the Institute for Computational Engineering and Sciences (ICES).

Authors' addresses: P. Bientinesi and R. A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: {pauldj,rvdg}@cs.utexas.edu; E. S. Quintana-Orti, Departamento de Ingieneria y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain; email: quintana@icc.uji.es.

specific architectures; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

General Terms: Algorithms, Design, Theory, Performance

Additional Key Words and Phrases: Application program interfaces, formal derivation, linear algebra, high-performance libraries

1. INTRODUCTION

The Formal Linear Algebra Methods Environment (FLAME) encompasses a methodology for deriving provably correct algorithms for dense linear algebra operations as well as an approach to representing (coding) the resulting algorithms [Gunnels et al. 2001; Quintana-Ortí and van de Geijn 2003; Bientinesi et al. 2005b]. Central to the philosophy underlying FLAME are the observations that one best reasons about the correctness of algorithms at a high level of abstraction and therefore algorithms should themselves be expressed at a high level of abstraction, and that codes that implement such algorithms should themselves use an API that captures this high level of abstraction. A key observation is that in reasoning about algorithms, intricate indexing is typically avoided and it is with the introduction of complex indexing that programming errors are often introduced and confidence in code is diminished. Thus a carefully designed API should avoid explicit indexing whenever possible. In this article we give such APIs for the MATLAB M-script and C programming languages [Moler et al. 1987; Kernighan and Ritchie 1978]. We also show the resulting MATLAB and C implementations to be part of a natural migratory path towards high-performance parallel implementation.

Our FLAME@lab, FLAME/C and FLAME/PLAPACK interfaces strive to allow algorithms to be presented in code so that the knowledge expressed in the algorithms is also expressed in the code. In particular, this knowledge is not obscured by intricate indexing. In a typical development, an initial FLAME@lab implementation gives the user the flexibility of MATLAB to test the algorithms designed using FLAME before going to a high-performance sequential implementation using the FLAME/C API, and the subsequent parallel implementation using the Parallel Linear Algebra Package (PLAPACK) [van de Geijn 1997; Baker et al. 1998; Alpatov et al. 1997]. In our experience, an inexperienced user can use these different interfaces to develop and test MATLAB and high-performance C implementations of an algorithm in less than an hour. An experienced user can perform this task in a matter of minutes, and can in addition implement a scalable parallel implementation in less than a day. This represents a significant reduction in effort relative to more traditional approaches to such library development [Anderson et al. 1992; Choi et al. 1992].

The FLAME approach to deriving algorithms often yields a large number of algorithms for a given linear algebra operation. Since the APIs given in this paper allow these algorithms to be easily captured in code, they enable the systematic creation of a repository of algorithms and their implementations. As part of our work, we have started to assemble such a repository, the FLAME Interface Repository (FIREsite).

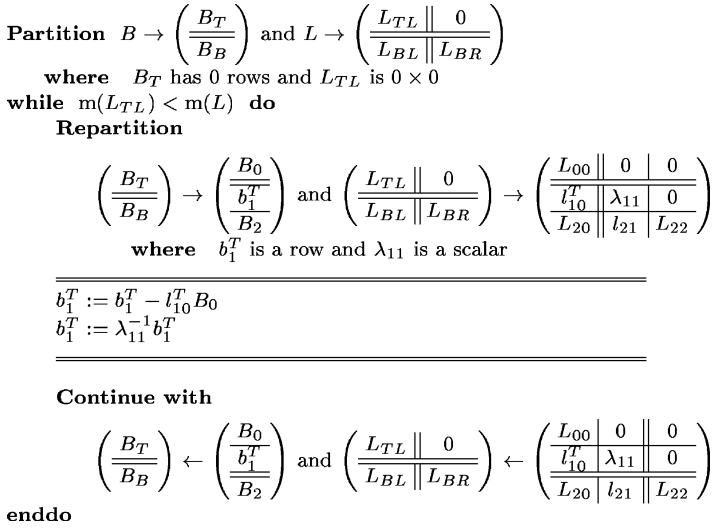


Fig. 1. Unblocked algorithm for triangular system solves (TRSM algorithm).

This article is organized as follows: In Section 2, we give an example of how we represent a broad class of linear algebra algorithms in our previous articles. The most important components of the FLAME@lab API are presented in Section 3. The FLAME/C API is given in Section 4. A discussion of how the developed algorithms, coded using the FLAME/C API, can be migrated to parallel code written in C is discussed in Section 5. Performance issues are discussed in Section 6. We discuss productivity issues and FIREsite in Section 7. A few concluding remarks are given in Section 8. In the electronic appendix, a listing of the most commonly used FLAME/C routines is given, as is a discussion regarding how to interface more traditional code with FLAME/C.

There are some repeated comments in Sections 3 and 4. Thus a reader can choose to skip the discussion of the FLAME@lab API in Section 3 or the FLAME/C API in Section 4 while fully benefiting from the insights in those sections. We assume the reader to have some experience with the MATLAB M-script and the C programming languages.

2. A TYPICAL DENSE LINEAR ALGEBRA ALGORITHM

In Bientinesi et al. [2005b] we introduced a methodology for the systematic derivation of provably correct algorithms for dense linear algebra algorithms. It is highly recommended that the reader become familiar with that article before proceeding with the remainder of this one. This section gives the minimal background in an attempt to make the present article self-contained.

The algorithms that result from the derivation process present themselves in a very rigid format. We illustrate this format in Figure 1, which gives an (unblocked) algorithm for the computation of $B := L^{-1}B$, where B is an $m \times n$ matrix and L is an $m \times m$ lower triangular matrix. This operation is often referred to as a triangular solve with multiple right-hand sides (TRSM). The presented algorithm was derived in Bientinesi et al. [2005b].

At the top of the loop body, it is assumed that different regions of the operands L and B have been used and/or updated in a consistent fashion. These regions are initialized by

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

Here T, B, L, and R stand for Top, Bottom, Left, and Right, respectively.

NOTE 1. *Of particular importance in the algorithm are the single and double lines used to partition and repartition the matrices. Double lines are used to demark regions in the matrices that have been used and/or updated in a consistent fashion. Another way of interpreting double lines is that they keep track of how far into the matrices the computation has progressed.*

Let $\hat{B}$ equal the original contents of B and assume that $\hat{B}$ is partitioned as B. At the top of the loop it is assumed that B_B contains the original contents $\hat{B}_B$ while B_T has been updated with the contents $L_{TL}^{-1}\hat{B}_T$. As part of the loop, the boundaries between these regions are moved one row and/or column at a time so that progress towards completion is made. This is accomplished by

Repartition

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$$
$$\textbf{where } b_1^T \text{ is a row and } \lambda_{11} \text{ is a scalar}$$
$$\vdots$$

Continue with

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right).$$

NOTE 2. *Single lines are introduced in addition to the double lines to demark regions that are involved in the update or used in the current step of the algorithm. Upon completion of the update, the regions defined by the double lines are updated to reflect that the computation has moved forward.*

NOTE 3. *We adopt the often-used convention where matrices, vectors, and scalars are denoted by upper-case, lower-case, and Greek letters, respectively* [Stewart 1973].

NOTE 4. *A row vector is indicated by adding a transpose to a vector, for example,* b_1^T *and* l_{10}^T.

The repartitioning exposes submatrices that must be updated before the boundaries can be moved. That update is given by

Partition $L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$ and $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$

where B_T has 0 rows and L_{TL} is 0×0

while $\mathrm{m}(L_{TL}) < \mathrm{m}(L)$ **do**

Determine block size b

Repartition

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

where $\mathrm{m}(B_1) = b$ and $\mathrm{m}(L_{11}) = \mathrm{n}(L_{11}) = b$

$B_1 := B_1 - L_{10}B_0$
$B_1 := L_{11}^{-1}B_1$

Continue with

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

enddo

Fig. 2. Blocked algorithm for triangular system solves (TRSM algorithm).

$b_1^T := b_1^T - l_{10}^T B_0$
$b_1^T := \lambda_{11}^{-1} b_1^T$

Finally, the desired result has been computed when L_{TL} encompasses all of L so that the loop continues until $\mathrm{m}(L_{TL}) < \mathrm{m}(L)$ becomes *false*. Here $m(X)$ returns the row dimension of matrix X.

NOTE 5. *We would like to claim that the algorithm in Figure* 1 *captures how one might naturally explain a particular algorithmic variant for computing the solution of a triangular linear system with multiple right-hand sides.*

NOTE 6. *The presented algorithm only requires one to use indices from the sets* $\{T, B\}$, $\{L, R\}$, *and* $\{0, 1, 2\}$.

For performance reasons, it is often necessary to formulate the algorithm as a *blocked* algorithm as illustrated in Figure 2. The performance benefit comes from the fact that the algorithm is rich in matrix multiplication, which allows processors with multi-level memories to achieve high performance [Dongarra et al. 1990, 1991; Anderson et al. 1992; Gunnels et al. 2001].

NOTE 7. *The algorithm in Figure* 2 *is implemented by the more traditional MATLAB code given in Figure* 3. *We claim that the introduction of indices to explicitly indicate the regions involved in the update complicates readability and reduces confidence in the correctness of the MATLAB implementation. Indeed, an explanation of the code inherently requires the drawing of a picture that captures the repartitioned matrices in Figure* 2. *In other words, someone experienced with*

```
[ m, n ] = size( B );
for i=1:mb:m
  b = min( mb, m-i+1 );
  B( i:i+b-1, : ) = B( i:i+b-1, : ) - ...
     L( i:i+b-1, 1:i-1 ) * B( 1:i-1, : );
  B( i:i+b-1, : ) = L( i:i+b-1, i:i+b-1 ) \ B( i:i+b-1, : );
end
```

Fig. 3. MATLAB implementation for blocked triangular system solves (TRSM algorithm in Figure 2). Here, `mb` is a parameter that determines the theoretical value for the block size and `b` is the actual block size.

MATLAB can easily translate the algorithm in Figure 2 *into the implementation in Figure* 3*. The converse is considerably more difficult.*

3. THE FLAME@LAB INTERFACE FOR MATLAB

In this section we introduce a set of MATLAB M-script functions that allow us to capture in code, the linear algebra algorithms presented in the format illustrated in the previous section. The idea is that by making the appearance of the code similar to the algorithms in Figures 1 and 2, the opportunity for the introduction of coding errors is reduced while simultaneously making the code more readable.

3.1 Bidimensional Partitionings

As illustrated in Figures 1 and 2, in stating a linear algebra algorithm, one may wish to partition a matrix as

$$\textbf{Partition}\ \ A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TL} \\ \hline A_{BL} & A_{BR} \end{array}\right)$$
$$\textbf{where}\ \ A_{TL} \text{ is } k \times k.$$

In the FLAME@lab API, we hide complicated indexing by using MATLAB matrices. Given a MATLAB matrix A, the following call creates one matrix for each of the four quadrants:

```
[ ATL, ATR,...
  ABL, ABR     ] = FLA_Part_2x2( A,...
                                 mb, nb, quadrant )
```

Purpose: Partition matrix A into four quadrants where the quadrant indicated by `quadrant` is `mb` $\times$ `nb`.

Here `quadrant` is a MATLAB string that can take on the values `'FLA_TL'`, `'FLA_TR'`, `'FLA_BL'`, and `'FLA_BR'` to indicate that `mb` and `nb` are the dimensions of the Top-Left, Top-Right, Bottom-Left, or Bottom-Right quadrant, respectively.

NOTE 8. *Invocation of the operation*

```
[ ATL, ATR,...
  ABL, ABR     ] = FLA_Part_2x2( A,...
                                 mb, nb, 'FLA_TL' )
```

in MATLAB creates four new matrices, one for each quadrant. Subsequent modifications of the contents of a quadrant therefore do not affect the original contents of the matrix. This is an important difference to consider with respect to the FLAME/C API introduced in Section 4, where the quadrants are views (references) into the original matrix, not copies of it!

As an example of the use of this routine, the translation of the algorithm fragment on the left results in the code on the right:

Partition $A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TL} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is $m_b \times n_b$

```
[ ATL, ATR,...
  ABL, ABR     ] = FLA_Part_2x2( A,...
                                 mb, nb, ...
                                 'FLA_TL' )
```

where the parameters `mb` and `nb` have values m_b and n_b, respectively. Examples of the use of this routine can also be found in Figures 4 and 5.

NOTE 9. *The above example stresses the fact that the formatting of the code can be used to help capture the algorithm in code. Clearly, some of the benefit of the API would be lost if in the example the code appeared as*

```
[ ATL, ATR, ABL, ABR  ] = FLA_Part_2x2( A, mb, nb, 'FLA_TL' )
```

since then, the left-hand side does not carry an intuitive image that `ATL,...,ABR` *are the corresponding blocks of a* 2×2 *partitioning.*

Also from Figures 1 and 2, we notice that it is useful to be able to take a 2×2 partitioning of a given matrix A and repartition it into a 3×3 partitioning so that the submatrices that need to be updated and/or used for computation can be identified. To support this, we introduce the call

```
[ A00, A01, A02,...
  A10, A11, A12,...
  A20, A21, A22     ] =  FLA_Repart_2x2_to_3x3( ATL, ATR,...
                                                ABL, ABR,...
                                                mb, nb, quadrant )
```

Purpose: Repartition a 2×2 partitioning of matrix `A` into a 3×3 partitioning where the `mb` $\times$ `nb` submatrix `A11` is split from the quadrant indicated by `quadrant`.

Here `quadrant` can again take on the values `'FLA_TL'`, `'FLA_TR'`, `'FLA_BL'`, and `'FLA_BR'` to indicate that the `mb` $\times$ `nb` submatrix `A11` is split from submatrix `ATL`, `ATR`, `ABL`, or `ABR`, respectively.

Thus,

Repartition

$$\left(\begin{array}{c||c} A_{TL} & A_{TL} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $m_b \times n_b$

```
function [ X ] = Trsm_llnn_unb_var1( L, B )

  [ LTL, LTR,...
    LBL, LBR     ] = FLA_Part_2x2( L,...
                                   0, 0, 'FLA_TL' );
  [ BT,...
    BB     ] = FLA_Part_2x1( B,...
                             0, 'FLA_TOP' );

  while( size( LTL, 1 ) < size( L, 1 ) )
    [ L00,  l01,      L02,...
      l10t, lambda11, l12t,...
      L20,  l21,      L22      ] = FLA_Repart_2x2_to_3x3( LTL, LTR,...
                                                          LBL, LBR,...
                                                          1, 1, 'FLA_BR' );
    [ B0,...
      b1t,...
      B2      ] = FLA_Repart_2x1_to_3x1( BT,...
                                         BB,...
                                         1, 'FLA_BOTTOM' );
%* ------------------------------------------------------------------- */
    b1t = b1t - l10t * B0;
    b1t = inv( lambda11 ) * b1t;
%* ------------------------------------------------------------------- */
    [ LTL, LTR,...
      LBL, LBR     ] = FLA_Cont_with_3x3_to_2x2( L00,  l01,      L02,...
                                                 l10t, lambda11, l12t,...
                                                 L20,  l21,      L22,...
                                                 'FLA_TL' );
    [ BT,...
      BB     ] = FLA_Cont_with_3x1_to_2x1( B0,...
                                           b1t,...
                                           B2,...
                                           'FLA_TOP' );
  end

  X = BT;
  return;
```

Fig. 4. FLAME implementation for unblocked triangular system solves (TRSM algorithm in Figure 1) using the FLAME@lab interface.

translates to the code

```
[ A00, A01, A02,...
  A10, A11, A12,...
  A20, A21, A22     ] =  FLA_Repart_2x2_to_3x3( ATL, ATR,...
                                                ABL, ABR,...
                                                mb, nb, 'FLA_BR' )
```

where the parameters `mb` and `nb` have values m_b and n_b, respectively. Other examples of the use of this routine can also be found in Figures 4 and 5.

NOTE 10. *Similarly to what is expressed in Note* 8*, the invocation of the operation*

```
function [ X ] = Trsm_llnn_blk_var1( L, B, mb )

  [ LTL, LTR,...
    LBL, LBR     ] = FLA_Part_2x2( L,...
                                   0, 0, 'FLA_TL' );
  [ BT,...
    BB     ] = FLA_Part_2x1( B,...
                             0, 'FLA_TOP' );

  while( size( LTL, 1 ) < size( L, 1 ) )
    b = min( mb, size( LBR, 1 ) );

    [ L00, L01, L02,...
      L10, L11, L12,...
      L20, L21, L22     ] = FLA_Repart_2x2_to_3x3( LTL, LTR,...
                                                   LBL, LBR,...
                                                   b, b, 'FLA_BR' );
    [ B0,...
      B1,...
      B2     ] = FLA_Repart_2x1_to_3x1( BT,...
                                        BB,...
                                        b, 'FLA_BOTTOM');
%* ------------------------------------------------------------------- */
    B1 = B1 - L10 * B0;
    B1 = Trsm_llnn_unb_var1( L11, B1 );
%* ------------------------------------------------------------------- */
    [ LTL, LTR,...
      LBL, LBR     ] = FLA_Cont_with_3x3_to_2x2( L00, L01, L02,...
                                                 L10, L11, L12,...
                                                 L20, L21, L22,...
                                                 'FLA_TL' );
    [ BT,...
      BB     ] = FLA_Cont_with_3x1_to_2x1( B0,...
                                           B1,...
                                           B2,...
                                           'FLA_TOP' );
  end

  X = BT;
  return;
```

Fig. 5. FLAME implementation for blocked triangular system solves (TRSM algorithm in Figure 2) using the FLAME@lab interface.

```
[ A00, A01, A02,...
  %A10, A11, A12,...
  %A20, A21, A22     ] =  FLA_Repart_2x2_to_3x3( ... )
```

creates nine new matrices `A00, A01, A02, ....`

NOTE 11. *Choosing variable names can further relate the code to the algorithm, as is illustrated by comparing*

$$\left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right) \quad \textit{and} \quad \begin{array}{lll} \texttt{L00}, & \texttt{l01} & \texttt{L02} \\ \texttt{l10t}, & \texttt{lambda11}, & \texttt{l12t} \\ \texttt{L20}, & \texttt{l21}, & \texttt{L22}, \end{array}$$

in Figures 1 *and* 4. *Although in the algorithm certain regions are identified as containing only zeroes, variables are needed to store those regions in the partitioning.*

Once the contents of the so-identified submatrices have been updated, the contents of ATL, ATR, ABL, and ABR must be updated to reflect that progress is being made, in terms of the regions indicated by the double lines. This moving of the double lines is accomplished by a call to

```
[ ATL, ATR,...
  ABL, ABR     ] = FLA_Cont_with_3x3_to_2x2( A00, A01, A02,...
                                             A10, A11, A12,...
                                             A20, A21, A22,...
                                             quadrant )
```

Purpose: Update the 2×2 partitioning of matrix A by moving the boundaries so that A11 is joined to the quadrant indicated by quadrant.

This time the value of quadrant ('FLA_TL', 'FLA_TR', 'FLA_BL', or 'FLA_BR') indicates to which quadrant the submatrix A11 is to be joined.

For example,

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TL} \\ \hline A_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

translates to the code

```
[ ATL, ATR,...
  ABL, ABR     ] = FLA_Cont_with_3x3_to_2x2( A00, A01, A02,...
                                             A10, A11, A12,...
                                             A20, A21, A22,...
                                             'FLA_TL' )
```

Further examples of the use of this routine can again be found in Figures 4 and 5.

3.2 Horizontal Partitionings

Similar to the partitioning into quadrants discussed above, and as illustrated in Figures 1 and 2, in stating a linear algebra algorithm, one may wish to partition a matrix as

Partition $A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$

where A_T has k rows.

For this, we introduce the call

```
[ AT,...
  AB     ] = FLA_Part_2x1( A,...
                           mb, side )
```

Purpose: Partition matrix `A` into a top and a bottom side where the side indicated by `side` has `mb` rows.

Here `side` can take on the values `'FLA_TOP'` or `'FLA_BOTTOM'` to indicate that `mb` is the row dimension of `AT` or `AB`, respectively.

Given that matrix `A` is already partitioned horizontally it can be repartitioned into three submatrices with the call

```
[ A0,...
  A1,...
  A2     ] = FLA_Repart_2x1_to_3x1( AT,...
                                    AB,...
                                    mb, side )
```

Purpose: Repartition a 2×1 partitioning of matrix `A` into a 3×1 partitioning where submatrix `A1` with `mb` rows is split from the bottom of `AT` or the top of `AB`, as indicated by `side`.

Here `side` can take on the values `'FLA_TOP'` or `'FLA_BOTTOM'` to indicate that submatrix `A1`, with `mb` rows, is partitioned from `AT` or `AB`, respectively.

Given a 3×1 partitioning of a given matrix `A`, the middle submatrix can be appended to either the first or last submatrix with the call

```
[ AT,...
  AB     ] = FLA_Cont_with_3x1_to_2x1( A0,...
                                       A1,...
                                       A2,...
                                       side )
```

Purpose: Update the 2×1 partitioning of matrix `A` by moving the boundaries so that `A1` is joined to the side indicated by `side`.

Examples of the use of the routines that deal with the horizontal partitioning of matrices can be found in Figures 4 and 5.

3.3 Vertical Partitionings

Finally, in stating a linear algebra algorithm, one may wish to partition a matrix as

Partition $A \rightarrow \left(A_L \middle\| A_R \right)$
where A_L has k columns .

For this we introduce the call

```
[ AL, AR ] = FLA_Part_1x2( A,...
                           int nb, int side )
```

Purpose: Partition matrix `A` into a left and a right side where the side indicated by `side` has `nb` columns.

and

```
[ A0, A1, A2 ] = FLA_Repart_1x2_to_1x3( AL, AR,...
                                        nb, side )
```

Purpose: Repartition a 1×2 partitioning of matrix `A` into a 1×3 partitioning where submatrix `A1` with `nb` columns is split from the right of `AL` or the left of `AR`, as indicated by `side`.

Here `side` can take on the values `'FLA_LEFT'` or `'FLA_RIGHT'`. Adding the middle submatrix to the first or last submatrix is now accomplished by a call to

```
[ AL, AR ] = FLA_Cont_with_1x3_to_1x2( A0, A1, A2,...
                                       side )
```

Purpose: Update the 1×2 partitioning of matrix `A` by moving the boundaries so that `A1` is joined to the side indicated by `side`.

3.4 Additional Routines

NOTE 12. *Interestingly enough, the routines described in this section for the MATLAB M-script language suffice to implement a broad range of algorithms encountered in dense linear algebra. So far, we have yet to encounter algorithms that cannot be elegantly described by partitioning into regions than can be indexed by the sets* $\{T, B\}$, $\{L, R\}$, $\{0, 1, 2\}$, $\{T, B\} \times \{L, R\}$, *and* $\{0, 1, 2\} \times \{0, 1, 2\}$. *However, there might be a potential use for a* 4×4 *partitioning in the future. Also, MATLAB provides a rich set of operations on matrices and vectors, which are needed to implement the updates to the exposed submatrices.*

4. THE FLAME/C INTERFACE FOR THE C PROGRAMMING LANGUAGE

It is easily recognized that the FLAME@lab codes given in the previous section will likely fall short of attaining peak performance. In particular, the copying that inherently occurs when submatrices are created and manipulated represents pure overhead. But then, generally people do not use MATLAB if they insist on attaining high performance. For that, they tend to code in C and link to high-performance libraries such as the Basic Linear Algebra Subprograms (BLAS) and the Linear Algebra Package (LAPACK) [Anderson et al. 1992; Lawson et al. 1979; Dongarra et al. 1988, 1990]. In this section we introduce a set of library routines that allow us to capture in C code, linear algebra algorithms presented in the format given in Section 2.

Again, the idea is that by making C code look similar to the algorithms in Figures 1 and 2, the opportunity for the introduction of coding errors is reduced. Readers familiar with MPI [Gropp et al. 1994; Snir et al. 1996], PETSc [Balay et al. 1996], and/or our own PLAPACK, will recognize the programming style,

object-based programming, as being very similar to that used by those (and other) interfaces. It is this style of programming that allows us to hide the indexing details much as MATLAB does. However, as we will see, a more substantial infrastructure must be provided in addition to the routines that partition and repartition matrix objects.

4.1 Initializing and Finalizing FLAME/C

Before using the FLAME/C environment, one must initialize with a call to

```
void FLA_Init( )
```

Purpose: Initialize FLAME/C.

If no more FLAME/C calls are to be made, the environment is exited by calling

```
void FLA_Finalize( )
```

Purpose: Finalize FLAME/C.

4.2 Linear Algebra Objects

The following attributes describe a matrix as it is stored in the memory of a computer:

—the datatype of the entries in the matrix, for example, `double` or `float`,
—m and n, the row and column dimensions of the matrix,
—the address where the data is stored, and
—the mapping that describes how the two-dimensional array is mapped to one-dimensional memory.

The following call creates an object (*descriptor* or *handle*) of type `FLA_Obj` for a matrix and creates space to store the entries in the matrix:

```
void FLA_Obj_create( int datatype, int m, int n, FLA_Obj *matrix )
```

Purpose: Create an object that describes an `m` $\times$ `n` matrix and create the associated storage array.

Valid datatype values include

`FLA_INT`, `FLA_DOUBLE`, `FLA_FLOAT`, `FLA_DOUBLE_COMPLEX`, and `FLA_COMPLEX`

for the obvious datatypes that are commonly encountered. The leading dimension of the array that is used to store the matrix is itself determined inside of this call.

NOTE 13. *For simplicity, we chose to limit the storage of matrices to use column-major storage. The leading dimension of a matrix can be thought of as the dimension of the array in which the matrix is embedded (which is often larger than the row-dimension of the matrix) or as the increment (in elements) required to address consecutive elements in a row of the matrix. Column-major*

storage is chosen to be consistent with Fortran, which is often still the choice of language for linear algebra applications. A C programmer should take this into account in case he needs to interface with the FLAME/C API.

FLAME/C treats vectors as special cases of matrices: an $n \times 1$ matrix or a $1 \times n$ matrix. Thus, to create an object for a vector x of n double-precision real numbers either of the following calls suffices:

```
FLA_Obj_create( FLA_DOUBLE, n, 1, &x );
FLA_Obj_create( FLA_DOUBLE, 1, n, &x );
```

Here `n` is an integer variable with value n and `x` is an object of type `FLA_Obj`.

Similarly, FLAME/C treats scalars as a 1×1 matrix. Thus, the following call is made to create an object for a scalar α:

```
FLA_Obj_create( FLA_DOUBLE, 1, 1, &alpha );
```

where `alpha` is an object of type `FLA_Obj`. A number of scalars occur frequently and are therefore predefined by FLAME/C:

`MINUS_ONE`, `ZERO`, and `ONE`.

If an object is created with `FLA_Obj_create` (or `FLA_Obj_create_conf_to`, given in the electronic appendix), a call to `FLA_Obj_free` is required to ensure that all space associated with the object is properly released:

```
void FLA_Obj_free( FLA_Obj *matrix )
```

Purpose: Free all space allocated to store data associated with `matrix`.

4.3 Inquiry Routines

In order to be able to work with the raw data, a number of inquiry routines can be used to access information about a matrix (or vector or scalar). The datatype and row and column dimensions of the matrix can be extracted by calling

```
int FLA_Obj_datatype( FLA_Obj matrix )
int FLA_Obj_length  ( FLA_Obj matrix )
int FLA_Obj_width   ( FLA_Obj matrix )
```

Purpose: Extract datatype, row, or column dimension of `matrix`, respectively.

The address of the array that stores the matrix and its leading dimension can be retrieved by calling

```
void *FLA_Obj_buffer( FLA_Obj matrix )
int FLA_Obj_ldim    ( FLA_Obj matrix )
```

Purpose: Extract address and leading dimension of `matrix`, respectively.

4.4 A Most Useful Utility Routine

Our approach to the implementation of algorithms for linear algebra operations starts with the careful derivation of provably correct algorithms. The stated philosophy is that if the algorithms are correct, and the API allows the algorithms to be coded so that the code reflects the algorithms, then the code will be correct as well.

Nonetheless, we single out one of the more useful routines in the FLAME/C library, which is particularly helpful for testing:

```
void FLA_Obj_show( char *string1, FLA_Obj A, char *format,
                   char *string2 )
```

Purpose: Print the contents of `A`.

In particular, the result of

```
FLA_Obj_show( "A =[", A, "%lf", "];" );
```

is similar to

```
A = [
< entries_of_A >
];
```

which can then be fed to MATLAB. This becomes useful when checking results against a MATLAB implementation of an operation.

4.5 An Example: Matrix-Vector Multiplication

We now give an example of how to use the calls introduced so far to write a simple driver routine that calls a routine that performs the matrix-vector multiplication $y = Ax$.

In Figure 6 we give the driver routine:

—**line 1:** FLAME/C program files start by including the `FLAME.h` header file.
—**line 5–6:** FLAME/C objects `A`, `x`, and `y`, which hold matrix A and vectors x and y, are declared to be of type `FLA_Obj`.
—**line 10:** Before any calls to FLAME/C routines can be made, the environment must be initialized by a call to `FLA_Init`.
—**line 12–13:** In our example, the user inputs the row and column dimension of matrix A.
—**line 15–17:** Descriptors are created for A, x, and y.
—**line 19–20:** The routine in Figure 7, described below, is used to fill A and x with values.
—**line 22:** Compute $y = Ax$ using the routine for performing that operation given in Figure 8.
—**line 24–26:** Print out the contents of A, x, and y.
—**line 28–30:** Free the created objects.
—**line 32:** Finalize FLAME/C.

```
1  #include "FLAME.h"
2
3  main()
4  {
5    FLA_Obj
6      A, x, y;
7    int
8      m, n;
9
10   FLA_Init( );
11
12   printf( "enter matrix dimensions m and n:" );
13   scanf( "%d%d", &m, &n );
14
15   FLA_Obj_create( FLA_DOUBLE, m, n, &A );
16   FLA_Obj_create( FLA_DOUBLE, m, 1, &y );
17   FLA_Obj_create( FLA_DOUBLE, n, 1, &x );
18
19   fill_matrix( A );
20   fill_matrix( x );
21
22   mv_mult( A, x, y );
23
24   FLA_Obj_show( "A = [", A, "%lf", "]" );
25   FLA_Obj_show( "x = [", x, "%lf", "]" );
26   FLA_Obj_show( "y = [", y, "%lf", "]" );
27
28   FLA_Obj_free( &A );
29   FLA_Obj_free( &y );
30   FLA_Obj_free( &x );
31
32   FLA_Finalize( );
33 }
```

Fig. 6. A simple C driver for matrix-vector multiplication.

A sample routine for filling A and x with data is given in Figure 7. The macro definition in line 3 is used to access the matrix A stored in array A using column-major ordering.

The routine in Figure 8 is itself a wrapper to the level 2 BLAS routine cblas_dgemv, a commonly available kernel for computing a matrix-vector multiplication that is part of the C interface to the legacy BLAS [BLAST Forum 2001]. In order to call this routine, which requires parameters describing the matrix, vectors, and scalars to be explicitly passed, all of the inquiry routines are required.

4.6 Views

Figures 1 and 2 illustrate the need for partitionings as

$$\textbf{Partition } A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TL} \\ \hline A_{BL} & A_{BR} \end{array}\right)$$
$$\textbf{where } A_{TL} \text{ is } k \times k.$$

```
 1  #include "FLAME.h"
 2
 3  #define BUFFER( i, j ) buff[ (j)*lda + (i) ]
 4
 5  void fill_matrix( FLA_Obj A )
 6  {
 7    int
 8      datatype, m, n, lda;
 9
10    datatype = FLA_Obj_datatype( A );
11    m        = FLA_Obj_length( A );
12    n        = FLA_Obj_width ( A );
13    lda      = FLA_Obj_ldim  ( A );
14
15    if ( datatype == FLA_DOUBLE ){
16      double *buff;
17      int    i, j;
18
19      buff = ( double * ) FLA_Obj_buffer( A );
20
21      for ( j=0; j<n; j++ )
22        for ( i=0; i<m; i++ )
23          BUFFER( i, j ) = i+j*0.01;
24    }
25    else FLA_Abort( "Datatype not yet supported", __LINE__, __FILE__ );
26  }
```

Fig. 7. A simple routine for filling a matrix.

In C we avoid complicated indexing by introducing the notion of a *view*, which is a **reference** into an existing matrix or vector. Given a descriptor `A` of a matrix A, the following call creates descriptors of the four quadrants:

```
void FLA_Part_2x2( FLA_Obj A, FLA_Obj *ATL, FLA_Obj *ATR,
                              FLA_Obj *ABL, FLA_Obj *ABR,
                   int mb, int nb, int quadrant )
```

Purpose: Partition matrix `A` into four quadrants where the quadrant indicated by `quadrant` is `mb` $\times$ `nb`.

Here `quadrant` can take on the values `FLA_TL`, `FLA_TR`, `FLA_BL`, and `FLA_BR` (defined in `FLAME.h`) to indicate that `mb` and `nb` specify the dimensions of the Top-Left, Top-Right, Bottom-Left, or Bottom-Right quadrant, respectively. Thus, the algorithm fragment on the left is translated into the code on the right

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TL} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is $m_b \times n_b$

```
FLA_Part_2x2( A, &ATL, /**/ &ATR,
                /* ************** */
                  &ABL, /**/&ABR,
               mb, nb, FLA_TL );
```

where parameters `mb` and `nb` have values m_b and n_b, respectively. Examples of the use of this routine can also be found in Figures 9 and 10.

```
#include "FLAME.h"
#include "cblas.h"

void mv_mult( FLA_Obj A, FLA_Obj x, FLA_Obj y )
{
  int
    datatype_A,    m_A, n_A, ldim_A,    m_x, n_y, inc_x,    m_y, n_y, inc_y;

  datatype_A = FLA_Obj_datatype( A );
  m_A        = FLA_Obj_length( A );
  n_A        = FLA_Obj_width ( A );
  ldim_A     = FLA_Obj_ldim  ( A );

  m_x        = FLA_Obj_length( x );       m_y        = FLA_Obj_length( y );
  n_x        = FLA_Obj_width ( x );       n_y        = FLA_Obj_width ( y );


  if ( m_x == 1 ) {
    m_x  = n_x;
    inc_x = FLA_Obj_ldim( x );
  }
  else inc_x = 1;

  if ( m_y == 1 ) {
    m_y  = n_y;
    inc_y = FLA_Obj_ldim( y );
  }
  else inc_y = 1;

  if ( datatype_A == FLA_DOUBLE ){
    double *buff_A, *buff_x, *buff_y;

    buff_A = ( double * ) FLA_Obj_buffer( A );
    buff_x = ( double * ) FLA_Obj_buffer( x );
    buff_y = ( double * ) FLA_Obj_buffer( y );

    cblas_dgemv( CblasColMaj, CblasNoTrans,
                 1.0, buff_A, ldim_A, buff_x, inc_x,
                 1.0, buff_y, inc_y );
  }
  else FLA_Abort( "Datatype not yet supported", __LINE__, __FILE__ );
}
```

Fig. 8. A simple matrix-vector multiplication routine. This routine is implemented as a wrapper to the BLAS routine `cblas_dgemv` for matrix-vector multiplication.

NOTE 14. *Invocation of the operation*

```
FLA_Part_2x2( A, &ATL, /**/ &ATR,
                /* ************** */
                  &ABL, /**/ &ABR,
              mb, nb, FLA_TL );
```

in C creates four views, one for each quadrant. Subsequent modifications of the contents of a view therefore affect the original contents of the matrix. This is an

```
#include "FLAME.h"

void Trsm_llnn_unb_var1( FLA_Obj L, FLA_Obj B )
{
  FLA_Obj     LTL, LTR,     L00,  l01,      L02,    BT,           B0,
              LBL, LBR,     l10t, lambda11, l12t,   BB,           b1t,
                            L20,  l21,      L22,                  B2;

  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 ) ){
    FLA_Repart_2x2_to_3x3( LTL, /**/ LTR,          &L00,  /**/ &l01,      &L02,
                         /* ************* */    /* ************************* */
                                /**/              &l10t, /**/ &lambda11, &l12t,
                           LBL, /**/ LBR,         &L20,  /**/ &l21,      &L22,
                           1, 1, /* lambda11 from */ FLA_BR );
    FLA_Repart_2x1_to_3x1( BT,                   &B0,
                          /**/                   /**/
                                                 &b1t,
                           BB,                   &B2,
                           1, /* length b1t from */ FLA_BOTTOM );
    /* ------------------------------------------------------------------- */

    FLA_Gemv( FLA_TRANSPOSE, MINUS_ONE, B0, l10t, ONE, b1t );
    FLA_Inv_scal( lambda11, b1t );

    /* ------------------------------------------------------------------- */
    FLA_Cont_with_3x3_to_2x2( &LTL, /**/ &LTR,      L00,  l01,      /**/ L02,
                                    /**/            l10t, lambda11, /**/ l12t,
                              /* ************** */ /* ************************ */
                                &LBL, /**/ &LBR,    L20,  l21,      /**/ L22,
                              /* lambda11 added to */ FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                                                       b1t,
                               /***/                   /**/
                                &BB,                   B2,
                               /* b1t  added to */ FLA_TOP );
  }
}
```

Fig. 9. FLAME/C implementation for unblocked triangular system solves (TRSM algorithm in Figure 1).

important difference to consider with respect to the FLAME@lab API introduced in Section 3, where the quadrants are copies of the original matrix!

NOTE 15. *The above example remarks that formatting the code as well as the careful introduction of comments helps in capturing the algorithm in code. Clearly, much of the benefit of the API would be lost if in the example the code appeared as*

```
FLA_Part_2x2( A, &ATL, &ATR, &ABL, &ABR, mb, nb, FLA_TL );
```

```
#include "FLAME.h"

void Trsm_llnn_var1_blk( FLA_Obj L, FLA_Obj B, 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 );

    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 );
    /* ------------------------------------------------------------------------ */

    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, L10, B0, ONE, B1 );
    Trsm_llnn_var1_unb( L11, B1 );

    /* ------------------------------------------------------------------------ */
    FLA_Cont_with_3x3_to_2x2( &LTL, /**/ &LTR,         L00, L01, /**/ L02,
                                   /**/                L10, L11, /**/ L12,
                             /* ************** */   /* ****************** */
                               &LBL, /**/ &LBR,         L20, L21, /**/ L22,
                               /* L11 added to */ FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                                                       B1,
                             /***/                    /**/
                               &BB,                    B2,
                               /* B1  added to */ FLA_TOP );
  }
}
```

Fig. 10. FLAME/C implementation for blocked triangular system solves (TRSM algorithm in Figure 2).

From Figures 1 and 2, we also realize the need for an operation that takes a 2×2 partitioning of a given matrix A and repartitions this into a 3×3 partitioning so that submatrices that need to be updated and/or used for computation can be identified. To support this, we introduce the call

```
void FLA_Repart_from_2x2_to_3x3
    ( FLA_Obj ATL, FLA_Obj ATR,   FLA_Obj *A00, FLA_Obj *A01, FLA_Obj *A02,
                                  FLA_Obj *A10, FLA_Obj *A11, FLA_Obj *A12,
      FLA_Obj ABL, FLA_Obj ABR,   FLA_Obj *A20, FLA_Obj *A21, FLA_Obj *A22,
      int mb, int nb, int quadrant )
```

Purpose: Repartition a 2×2 partitioning of matrix `A` into a 3×3 partitioning where `mb` $\times$ `nb` submatrix `A11` is split from the quadrant indicated by `quadrant`.

Here `quadrant` can again take on the values `FLA_TL`, `FLA_TR`, `FLA_BL`, and `FLA_BR` to indicate that `mb` $\times$ `nb` submatrix `A11` is split from submatrix `ATL`, `ATR`, `ABL`, or `ABR`, respectively.

Thus,

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TL} \\ \hline A_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $m_b \times n_b$,

is captured in the code

```
FLA_Repart_from_2x2_to_3x3( ATL, ATR,     &A00, /**/ &A01, &A02,
                                        /* ******************** */
                                          &A10, /**/ &A11, &A12,
                           ABL, ABR,      &A20, /**/ &A21, &A22,
                           mb, nb, FLA_BR );
```

where parameters `mb` and `nb` have values m_b and n_b, respectively. Others examples of the use of this routine can also be found in Figures 9 and 10.

NOTE 16. *The calling sequence of* `FLA_Repart_from_2x2_to_3x3` *and related calls is a testimony to throwing out the convention that input parameters should be listed before output parameters or vice versa, as well as to careful formating. It is specifically by mixing input and output parameters that the repartitioning in the algorithm can be elegantly captured in code.*

NOTE 17. *Chosing variable names can further relate the code to the algorithm, as is illustrated by comparing*

$$\left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right) \text{ and }$$

```
   L00,  /**/ l01,      L02,
/* ************************ */
   l10t, /**/ lambda11, l12t,
   L20,  /**/ l21,      L22, ...
```

in Figures 1 *and* 9.

Once the contents of the corresponding views have been updated, the descriptions of A_{TL}, A_{TL}, A_{BL}, and A_{BR} must be updated to reflect that progress is being made, in terms of the regions identified by the double lines. Moving the double lines is achieved by a call to

```
void FLA_Cont_with_3x3_to_2x2
     ( FLA_Obj *ATL, FLA_Obj *ATR,   FLA_Obj A00, FLA_Obj A01, FLA_Obj A02,
                                     FLA_Obj A10, FLA_Obj A11, FLA_Obj A12,
       FLA_Obj *ABL, FLA_Obj *ABR,   FLA_Obj A20, FLA_Obj A21, FLA_Obj A22,
       int quadrant )
```

Purpose: Update the 2×2 partitioning of matrix `A` by moving the boundaries so that `A11` is joined to the quadrant indicated by `quadrant`.

Here the value of `quadrant` (`FLA_TL`, `FLA_TR`, `FLA_BL`, or `FLA_`*BR*) specifies that the quadrant submatrix `A11` is to be joined.

For example,

Continue with

$$\left(\begin{array}{c||c} A_{TL} & A_{TL} \\ \hline\hline A_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

translates to the code

```
FLA_Cont_with_3x3_to_2x2( &ATL, &ATR,       A00, A01, /**/ A02,
                                            A10, A11, /**/ A12,
                                          /* ***************** */
                          &ABL, &ABR,       A20, A21, /**/ A22,
                          FLA_TL );
```

Further examples of the use of this routine can again be found in Figures 9 and 10.

Similarly, a matrix can be partitioned horizontally into two submatrices with the call

```
void FLA_Part_2x1( FLA_Obj A, FLA_Obj *AT,
                              FLA_Obj *AB,
                   int mb, int side )
```

Purpose: Partition matrix `A` into a top and bottom side where the side indicated by `side` has `mb` rows.

Here `side` can take on the values `FLA_TOP` or `FLA_BOTTOM` to indicate that `mb` indicates the row dimension of `AT` or `AB`, respectively.

Given that matrix A is already partitioned horizontally it can be repartitioned into three submatrices with the call

```
void FLA_Repart_from_2x1_to_3x1( FLA_Obj AT,   FLA_Obj *A0,
                                               FLA_Obj *A1,
                                 FLA_Obj AB,   FLA_Obj *A2,
                                 int mb, int side )
```

Purpose: Repartition a 2×1 partitioning of matrix `A` into a 3×1 partitioning where submatrix `A1` with `mb` rows is split from the side indicated by `side`.

Here `side` can take on the values `FLA_TOP` or `FLA_BOTTOM` to indicate that `mb` submatrix `A1` is partitioned from `AT` or `AB`, respectively.

Given a 3×1 partitioning of a given matrix A, the middle submatrix can be appended to either the first or last submatrix with the call

```
void FLA_Cont_with_3x1_to_2x1( FLA_Obj *AT,  FLA_Obj A0,
                                             FLA_Obj A1,
                               FLA_Obj *AB,  FLA_Obj A2,
                               int side )
```

Purpose: Update the 2×1 partitioning of matrix `A` by moving the boundaries so that `A1` is joined to the side indicated by `side`.

Examples of the use of the routine that deals with the horizontal partitioning of matrices can be found in Figures 9 and 10.

Finally, a matrix can be partitioned and repartitioned vertically with the calls

```
void FLA_Part_1x2( FLA_Obj A, FLA_Obj *AL, FLA_Obj *AR,
                   int nb, int side )
```

Purpose: Partition matrix `A` into a left and right side where the side indicated by `side` has `nb` columns.

and

```
void FLA_Repart_from_1x2_to_1x3
       ( FLA_Obj AL,               FLA_Obj AR,
         FLA_Obj *A0, FLA_Obj *A1, FLA_Obj *A2,
         int nb, int side )
```

Purpose: Repartition a 1×2 partitioning of matrix `A` into a 1×3 partitioning where submatrix `A1` with `nb` columns is split from the side indicated by `side`.

Here `side` can take on the values `FLA_LEFT` or `FLA_RIGHT`. Adding the middle submatrix to the first or last is now accomplished by a call to

```
void FLA_Cont_with_1x3_to_1x2
       ( FLA_Obj *AL,            FLA_Obj *AR,
         FLA_Obj A0, FLA_Obj A1, FLA_Obj A2,
         int side )
```

Purpose: Update the 1×2 partitioning of matrix `A` by moving the boundaries so that `A1` is joined to the side indicated by `side`.

4.7 Computational Kernels

All operations described in the last subsection hide the details of indexing in the linear algebra objects. To compute with and/or update data associated with a linear algebra object, one calls subroutines that perform the desired operations.

Such subroutines typically take one of three forms:

—subroutines coded using the FLAME/C interface (including, possibly, a recursive call),

—subroutines coded using a more traditional coding style, or

—wrappers to highly optimized kernels.

These are actually only three points on a spectrum of possibilities, since one can mix these techniques.

A subset of currently supported operations is given in the electronic appendix to this article. Here, we discuss how to create subroutines that compute these operations. For additional information on supported functionality, please visit the webpage given at the end of this article or [Gunnels and van de Geijn 2001a].

4.7.1 *Subroutines Coded Using the FLAME/C Interface.* The subroutine itself could be coded using the FLAME approach to deriving algorithms [Bientinesi et al. 2005b] and the FLAME/C interface described in this section.

For example, the implementation in Figure 10 of the blocked algorithm given in Figure 2 requires the update $B_1 := L_{11}^{-1} B_1$, which can be implemented by a call to the unblocked algorithm in Figure 9.

4.7.2 *Subroutine Coded Using a More Traditional Coding Style.* There is an overhead for the abstractions that we introduce to hide indexing. For implementations of blocked algorithms, this overhead is amortized over a sufficient amount of computation so that it is typically not of much consequence. (In the case of the algorithm in Figure 2 when B is $m \times n$, the indexing overhead is $O(m/b)$ while the useful computation is $O(m^2 n)$.) However, for unblocked algorithms or algorithms that operate on vectors, the relative cost is more substantial. In this case, it may become beneficial to code the subroutine using a more traditional style that exposes indices. For example, the operation

```
FLA_Inv_scal( lambda11, b1t );
```

can be implemented by the subroutine in Figure 11. (It is probably more efficient to instead implement it by calling `cblas_dscal` or the equivalent BLAS routine for the appropriate datatype.)

NOTE 18. *Even when a routine is ultimately implemented using more traditional code, it is beneficial to incorporate the FLAME/C code as comments for clarification.*

4.7.3 *Wrappers to Highly Optimized Kernels.* A number of matrix and/or vector operations have been identified to be frequently used by the linear algebra community. Many of these are part of the BLAS. Since highly optimized implementations of these operations are supported by widely available library

```
#include "FLAME.h"

void FLA_Inv_scal( FLA_Obj alpha, FLA_Obj x )
{
  int datatype_alpha, datatype_x, n_x, inc_x, i;
  double *buffer_alpha, *buffer_x, recip_alpha;

  datatype_alpha = FLA_Obj_datatype( alpha );
  datatype_x     = FLA_Obj_datatype( x );

  if (( datatype_alpha == FLA_DOUBLE ) &&
      ( datatype_x     == FLA_DOUBLE )) {

    n_x   = FLA_Obj_length( x );

    if ( n_x == 1 ){
      n_x = FLA_Obj_width( x );
      inc_x = FLA_Obj_ldim( x );
    }
    else inc_x = 1;

    buffer_alpha = ( double * ) FLA_Obj_buffer( alpha );
    buffer_x     = ( double * ) FLA_Obj_buffer( x );

    recip_alpha = 1.0 / *buffer_alpha;

    for ( i=0; i<n_x; i++ )
      *buffer_x++ *= recip_alpha;

    /* For BLAS based implementation, comment out above loop
       and uncomment the call to cblas_dscal below */

    /* cblas_dscal( n_x, recip_alpha, buffer_x, inc_x ); */
  }
  else FLA_Abort( "datatype not yet supported", __LINE__, __FILE__ );
}
```

Fig. 11. Sample implementation of the scaling routine `FLA_Inv_scal`.

implementations, it makes sense to provide a set of subroutines that are simply wrappers to the BLAS. An example of this is given in Figure 8.

5. FROM FLAME@LAB TO FLAME/C TO PLAPACK

As mentioned, we view the FLAME@lab and FLAME/C interfaces as tools on a migratory path that starts with the specification of the operation to be performed, after which the FLAME derivation process can be used to systematically derive a family of algorithms for computing the operation, followed by an initial implementation with FLAME@lab, a high-performance implementation with FLAME/C, and finally a parallel implementation with a FLAME/C-like extension of the PLAPACK interface.

5.1 Cases Where FLAME@lab is Particularly Useful

Since algorithms can clearly be directly translated to FLAME/C, the question of the necessity for the FLAME@lab API arises. As is well known, MATLAB-like

environments are extremely powerful interactive tools for manipulating matrices and investigating algorithms; interactivity is probably the key feature, allowing the user to dramatically speed up the design of procedures such as input generation and output analysis.

The authors have had the chance to exploit the FLAME@lab API in a number of research topics:

—In Quintana-Ortí and van de Geijn [2003], the interface was used to investigate the numerical stability properties of algorithms derived for the solution of the triangular Sylvester equation.

—In an ongoing study, we are similarly using it for the analysis of the stability of different algorithms for inverting a triangular matrix. Several algorithms exist for this operation. We derived them by using the FLAME methodology and implemented them with FLAME@lab. For each variant, measurements of different forms of residuals and forward errors had to be made [Higham 2002]. As part of the study, the input matrices needed to be chosen with extreme care; often they are results from some other operation, such as the `lu` function in MATLAB (which produces an LU factorization of a given matrix).

For these kinds of investigative studies, high performance is not required. It is the interactive nature of tools as MATLAB that is especially useful.

5.2 Moving on to FLAME/C

Once derived algorithms have been implemented and tested with FLAME@lab, the transition to a high-performance implementation using the FLAME/C API is direct, requiring (consultation of the appropriate documentation and) the translation for the operations in the loop body to calls to subroutines with the functionality of the BLAS.

The most significant difference between the FLAME/C and FLAME@lab APIs is that for the FLAME/C interface, the partitioning routines return views (i.e., references) into the matrix. Thus, any subsequent modification of the view results in a modification of the original contents of the matrix. The use of views in the FLAME/C API avoids much of the unnecessary data copying that occurs in the FLAME@lab API, possibly leading to a higher-performance implementation. It is possible to call C routines from MATLAB, and we have implemented such an interface. This could allow one to benefit from the interactive environment MATLAB provides, while retaining most of the performance benefits of coding subroutines in C.

5.3 And Finally the Parallel Implementation

While the PLAPACK API already hides details of indexing by using objects, and to a large degree inspired the FLAME/C API, the notion of tracking all submatrices of the matrices involved in the computation as FLAME/C does is new. Specifically, the routines `FLA_Repart_...` and `FLA_Cont_with_...` were not part of the original PLAPACK API. As part of our project, we have now added similar routines to the PLAPACK API. An implementation using PLAPACK for TRSM is given in Figure 12. In essence, a parallel implementation can be created

```
#include "PLA.h"

void PLA_Trsm_llnn_var1_blk( PLA_Obj L, PLA_Obj B, int nb_alg )
{
  PLA_Obj
    LTL=NULL, LTR=NULL,  L00=NULL, L01=NULL, L02=NULL,  BT=NULL, B0=NULL,
    LBL=NULL, LBR=NULL,  L10=NULL, L11=NULL, L12=NULL,  BB=NULL, B1=NULL,
                         L20=NULL, L21=NULL, L22=NULL,           B2=NULL,
    MINUS_ONE=NULL, ZERO=NULL, ONE=NULL;
  int         b;

  PLA_Create_constants_conf_to( L, &MINUS_ONE, &ZERO, &ONE );

  PLA_Part_2x2( L,  &LTL, /**/ &LTR,
                /* ************** */
                  &LBL, /**. &LBR,   0, 0,     /* submatrix */ PLA_TL );
  PLA_Part_2x1( B,  &BT,
                 /***/
                  &BB,               0, /* length submatrix */ PLA_TOP );

  while ( PLA_Obj_length( LTL ) < PLA_Obj_length( L ) ){
    b = min( PLA_Obj_length( LBR ), nb_alg );
    PLA_Repart_2x2_to_3x3( LTL, /**/ LTR,         &L00, /**/ &L01, &L02,
                         /* ************* */    /* ********************* */
                                /**/              &L10, /**/ &L11, &L12,
                           LBL, /**/ LBR,         &L20, /**/ &L21, &L22,
                           b, b, /* L11 from */ PLA_BR );
    PLA_Repart_2x1_to_3x1( BT,                    &B0,
                          /**/                    /**/
                                                  &B1,
                           BB,                    &B2,
                           b, /* length B1 from */ PLA_BOTTOM );
    /* ------------------------------------------------------------------------ */

    PLA_Gemm( PLA_NO_TRANSPOSE, PLA_NO_TRANSPOSE, MINUS_ONE, L10, B0, ONE, B1 );
    PLA_Trsm_llnn_var1_unb( L11, B1 );

    /* ------------------------------------------------------------------------ */
    PLA_Cont_with_3x3_to_2x2( &LTL, /**/ &LTR,         L00, L01, /**/ L02,
                                    /**/               L10, L11, /**/ L12,
                            /* ************** */   /* ****************** */
                              &LBL, /**/ &LBR,         L20, L21, /**/ L22,
                              /* L11 added to */ PLA_TL );
    PLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                                                       B1,
                             /***/                    /**/
                              &BB,                     B2,
                              /* B1  added to */ PLA_TOP );
  }
  PLA_Obj_free( &LTL );
    [ : ]
  PLA_Obj_free( &ONE );
}
```

Fig. 12. FLAME/PLAPACK implementation for blocked triangular system solves (TRSM algorithm in Figure 10).

by replacing FLAME.h with PLA.h and all prefixes FLA_ with PLA_. In PLAPACK, objects are defined as pointers to structures that are dynamically allocated. As a result, the declarations are somewhat different when compared to the FLAME/C code in Figure 10. Furthermore, these objects so allocated must be freed at the end of the routine. Finally, the constants MINUS_ONE, ZERO, and ONE must be created in each new routine. These idiosyncrasies suggest that it is time to update the PLAPACK API to become closer to the FLAME API.

In addition to attaining performance by casting computation as much as possible in terms of matrix-matrix operations (blocked algorithms), a parallel implementation requires careful assignment of data and work to individual processors. Clearly, the FLAME/C interface does not capture this, nor does the most trivial translation of FLAME/C to FLAME/PLAPACK. It is here where the full PLAPACK API allows the user to carefully manipulate the data and the operations, while still coding at a high level of abstraction. This manipulation is relatively systematic. Indeed, the Broadway compiler can to some degree automate this process [Guyer and Lin 1999, 2000a, 2000b; Guyer et al. 2001]. Also, an automated system for directly translating algorithms such as those given in Section 2 to optimized PLAPACK code has been prototyped [Gunnels 2001].

Further details regarding parallel implementations go beyond the scope of this article.

5.4 MATLAB to Parallel Implementations

In some sense, our work answers the question of how to generate parallel implementations from algorithms coded in MATLAB M-script [Moler et al. 1987]. For the class of problems to which this approach applies, the answer is to start with the algorithm, and to create APIs that can target MATLAB, C, or parallel architectures.

6. PERFORMANCE

In a number of articles that were already mentioned in the introduction we have shown that the FLAME/C API can be used to attain high performance for implementations of a broad range of linear algebra operations. Thus, we do not include a traditional performance section. Instead, we discuss some of the issues.

Conventional wisdom used to dictate that raising the level of abstraction at which one codes adversely impacts the performance of the implementation. We, like others, disagree for a number of reasons:

—By raising the level of abstraction, more ambitious algorithms can be implemented, which can achieve higher performance [Gunnels et al. 2001; Quintana-Ortí and van de Geijn 2003; Gunnels and van de Geijn 2001b; Bientinesi et al. 2002; Alpatov et al. 1997; van de Geijn 1997].

One can, of course, argue that these same algorithms can also be implemented at a lower level of abstraction. While this is true for individual operations, implementing entire libraries at a low level of abstraction greatly increases the effort required to implement, maintain, and verify correctness.

—Once implementations are implemented with an API at a high level of abstraction, components can be selectively optimized at a low level of abstraction. We learn from this that the API must be designed to easily accommodate this kind of optimization, as is also discussed in Section 4.7.

—Recent compiler technology [Guyer and Lin 1999, 2000a, 2000b; Guyer et al. 2001] allows library developers to specify dependencies between routines at a high level of abstraction, which allows compilers to optimize between layers of libraries, automatically achieving the kinds of optimizations that would otherwise be performed by hand.

—Other situations in which abstraction offers the opportunity for higher performance include several mathematical libraries and C++ optimization techniques as well. For example, PMLP [Birov et al. 1998] uses C++ templates to support many different storage formats, thereby decoupling storage format from algorithmic correctness in classes of sparse linear algebra, thus allowing this degree of freedom to be explored for optimizing performance. Also, PMLP features operation sequences and non-blocking operations in order to allow scheduling of mathematical operations asynchronously from user threads. Template meta-programming and expression templates support concepts including compile-time optimizations involving loop fusion, expression simplification, and removal of unnecessary temporaries; these allow C++ to utilize fast kernels while removing abstraction barriers between kernels, and further abstraction barriers between sequences of user operations (systems include Blitz++ [Veldhuizen 2001]). These techniques, in conjunction with an appropriate FLAME-like API for C++, should allow our algorithms to be expressed at a high level of abstraction without compromising performance.

NOTE 19. *The lesson to be learned is that by raising the level of abstraction, a high degree of confidence in the correctness of the implementation can be achieved while more aggressive optimizations, by hand or by a compiler, can simultaneously be facilitated.*

7. PRODUCTIVITY AND THE FLAME INTERFACE REPOSITORY (FIRE)

In the abstract and introduction of this article, we make claims regarding the impact of the presented approach on productivity. In this section, we narrate a few experiences.

7.1 Sequential Implementation of Algorithms for the Triangular Sylvester Equation

A clear demonstration that the FLAME derivation process, in conjunction with the FLAME APIs, can be used to quickly generate new algorithms and implementations for non-trivial operations came in the form of a family of algorithms for the triangular Sylvester equations. Numerous previously unknown high-performance algorithms were derived in a matter of hours, and implemented using the FLAME/C API in less than a day. In response to the submitted related article, referees requested that the numerical properties of the resulting implementations be investigated. In an effort to oblige, the FLAME@lab

interface was created, and numerical experiments were performed with the aid of the MATLAB environment. The resulting article has now appeared [Quintana-Ortí and van de Geijn 2003].

7.2 Parallel Implementation of the Reduction to Tridiagonal Form

As part of an effort to parallelize the Algorithm of Multiple Relatively Robust Representations (MRRR) for dense symmetric eigenproblems, a parallel implementation of the reduction to tridiagonal form via Householder transformations was developed using PLAPACK [Bientinesi et al. 2005a]. First, a careful description of the algorithms was created, in the format presented in Section 2. Next, a FLAME@lab implementation was created, followed by a FLAME/C implementation. Finally, the sequential code was ported to PLAPACK. The entire development of this implementation from start to finish took about a day of the time of two of the authors.

7.3 Undergraduate and Graduate Education

Before the advent of FLAME@lab and FLAME/C, projects related to the high-performance implementation of linear algebra algorithms required students to code directly in terms of BLAS calls with explicit indexing into arrays. Much more ambitious projects can now be undertaken by less experienced students since the most difficult component of the code, the indexing, has been greatly simplified.

7.4 Assembling the FLAME Interface REpository (FIRE)

As part of undergraduate and graduate courses at UT-Austin, students have been generating algorithms and implementations for a broad spectrum of linear algebra operations. An undergraduate in one of these classes, Minhaz Khan, took it upon himself to systematically assemble many of these implementations in the FLAME Interface REpository (FIRE). To date, hundreds of implementations of dozens of algorithms have been catalogued, almost half single-handedly by this student. After some experience was gained, he reported being able to derive, prove correct, implement, and test algorithms at a rate of about seven minutes per algorithm for BLAS-like operations involving several triangular matrices.[1]

8. CONCLUSION

In this article, we have presented simple APIs for implementing linear algebra algorithms using the MATLAB M-script and C programming languages. In isolation, these interfaces illustrate how raising the level of abstraction at which one codes allows one to avoid intricate indexing in the code, which reduces the opportunity for the introduction of errors and raises the confidence in the correctness of the code. In combination with our formal derivation methodology, the APIs can be used to implement algorithms derived using that methodology

[1]These operations are similar to those supported by the LAPACK auxiliary routines DLAUUM and DLAUU2.

so that the proven correctness of those algorithms translates to a high degree of confidence in the implementation.

We want to emphasize that the presented APIs are merely meant to illustrate the issues. Similar interfaces for the Fortran, C++, and other languages are easily defined, allowing special features of those languages to be used to raise even further the level of abstraction at which one codes.

Finally, an increasing number of linear algebra operations have been captured with our formal derivation methodology. This set of operations includes, to name but a few, the complete levels 1, 2, and 3 BLAS factorization operations such as the LU and QR (with and without pivoting), reduction to condensed forms, and linear matrix equations arising in control. An ever-growing collection of linear algebra operations written using the FLAME@lab and FLAME/C interfaces can be found at the URI given below.

FURTHER INFORMATION

For further information on the FLAME project and to download the FLAME@lab or FLAME/C interface, visit

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

The FIREsite repository is being maintained at

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

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

An ever-growing number of people have contributed to the methodology that underlies the Formal Linear Algebra Methods Environment. These include

—UT-Austin: Brian Gunter, Mark Hinga, Thierry Joffrain, Minhaz Khan, Tze Meng Low, Dr. Margaret Myers, Vinod Valsalam, Serita Van Groningen, and Field Van Zee.

—IBM's T.J. Watson Research Center: Dr. John Gunnels and Dr. Fred Gustavson.

—Intel: Dr. Greg Henry.

—University of Alabama at Birmingham: Prof. Anthony Skjellum and Wenhao Wu.

In addition, numerous students in undergraduate and graduate courses on high-performance computing at UT-Austin have provided valuable feedback.

Finally, we would like to thank the referees for their valuable comments that helped to improve the contents of this article.

REFERENCES

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

ANDERSON, E., BAI, Z., DEMMEL, J., DONGARRA, J. E., DUCROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A. E., OSTROUCHOV, S., AND SORENSEN, D. 1992. *LAPACK Users' Guide*. SIAM, Philadelphia.

BAKER, G., GUNNELS, J., MORROW, G., RIVIERE, B., AND VAN DE GEIJN, R. 1998. PLAPACK: High performance through high level abstraction. In *Proceedings of ICPP98*. IEEE Computer Society Press.

BALAY, S., GROPP, W., MCINNES, L. C., AND SMITH, B. 1996. PETSc 2.0 users manual. Tech. Rep. ANL-95/11, Argonne National Laboratory. Oct.

BIENTINESI, P., DHILLON, I., AND VAN DE GEIJN, R. 2005a. A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations. *SIAM J. Sci. Comput.*. To appear.

BIENTINESI, P., GUNNELS, J. A., GUSTAVSON, F. G., HENRY, G. M., MYERS, M. E., QUINTANA-ORTI, E. S., AND VAN DE GEIJN, R. A. 2002. The science of programming high-performance linear algebra libraries. In *Proceedings of Performance Optimization for High-Level Languages and Libraries (POHLL-02), a workshop in conjunction with the 16th Annual ACM International Conference on Supercomputing (ICS'02)*.

BIENTINESI, P., GUNNELS, J. A., MYERS, M. E., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2005b. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.* (March).

BIROV, L., PURKAYASTHA, A., SKJELLUM, A., DANDASS, Y., AND BANGALORE, P. V. 1998. PMLP home page. `http://www.erc.msstate.edu/labs/hpcl/pmlp`.

BLAST FORUM. 2001. *Basic Linear Algebra Suprograms Technical (BLAST) Forum Standard—Annex B*.

CHOI, J., DONGARRA, J. J., POZO, R., AND WALKER, D. W. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society Press, 120–127.

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

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

DONGARRA, J. J., DUFF, I. S., SORENSEN, D. C., AND VAN DER VORST, H. A. 1991. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA.

GROPP, W., LUSK, E., AND SKJELLUM, A. 1994. *Using MPI*. The MIT Press.

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

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

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

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

GUYER, S. Z., BERGER, E., AND LIN, C. 2001. Customizing software libraries for performance portability. In *10th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM.

GUYER, S. Z. AND LIN, C. 1999. An annotation language for optimizing software libraries. In *Second Conference on Domain Specific Languages*. ACM Press, 39–52.

GUYER, S. Z. AND LIN, C. 2000a. *Broadway: A Software Architecture for Scientific Computing*. Kluwer Academic Press, 175–192.

GUYER, S. Z. AND LIN, C. 2000b. Optimizing the use of high performance software libraries. In *Lecture Notes in Computer Sciences*, vol. 2017. Springer-Verlag, Berlin, 227–243.

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

KERNIGHAN, B. AND RITCHIE, D. 1978. *The C programming language*. Prentice-Hall, Englewood Cliffs, NJ.

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

MOLER, C., LITTLE, J., AND BANGERT, S. 1987. *Pro-Matlab, User's Guide*. The Mathworks, Inc.

QUINTANA-ORTÍ, E. S. AND VAN DE GEIJN, R. A. 2003. Formal derivation of algorithms for the triangular Sylvester equation. *ACM Trans. Math. Soft. 29*, 2, 218–243.

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

STEWART, G. W. 1973. *Introduction to Matrix Computations*. Academic Press, Orlando, Florida.

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

VELDHUIZEN, T. 2001. Blitz++ user's guide. URL: `http://oonumerics.org/blitz/`.

Received September 2003; revised April and October 2004; accepted October 2004

Accumulating Householder Transformations, Revisited

THIERRY JOFFRAIN and TZE MENG LOW
The University of Texas at Austin
ENRIQUE S. QUINTANA-ORTÍ
Universidad Jaume I
and
ROBERT VAN DE GEIJN and FIELD G. VAN ZEE
The University of Texas at Austin

A theorem related to the accumulation of Householder transformations into a single orthogonal transformation known as the compact WY transform is presented. It provides a simple characterization of the computation of this transformation and suggests an alternative algorithm for computing it. It also suggests an alternative transformation, the UT transform, with the same utility as the compact WY Transform which requires less computation and has similar stability properties. That alternative transformation was first published over a decade ago but has gone unnoticed by the community.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra; G.4 [**Mathematical Software**]: *Efficiency*

General Terms: Algorithms; Performance

Additional Key Words and Phrases: Linear algebra, Householder transformation, compact WY transform, QR factorization

1. INTRODUCTION

Given a nonzero vector $u \in \mathbb{R}^m$, a *Householder transformation* (or *reflector*) is defined by $H = I - \frac{uu^T}{\tau}$, where I denotes the (square) identity matrix and

This research was partially sponsored by NSF grants ACI-0305163 and CCF-0342369 and an equipment donation from Hewlett-Packard. Dr. Quintana-Orti was partially supported by a J. Tinsley Oden Visiting Research Fellowship from the UT-Austin Institute for Computational Engineering and Sciences. Access to the NEC SX-6 server was arranged by NEC Solutions (America), Inc.
Authors' addresses: T. Joffrain, T. M. Low, R. Van de Geijn, and F. G. Van Zee, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: {joffrain,itm,rvdg,field}@cs.utexas.edu; E. S. Quintana-Ortí, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, Campus Riu Sec, 12.071—Castellón, Spain; email: quintana@icc.uji.es.

$\tau = \frac{u^T u}{2}$ [Householder 1958]. It is an orthogonal matrix ($H^T H = HH^T = I$) and its own transpose ($H^T = H$). This transformation has wide application in the solution of linear least-squares problems, the computation of orthonormal bases, and the solution of the algebraic eigenvalue problem.

Two transforms that capture the action of multiple Householder transformations and cast it in terms of high-performance matrix-matrix products were proposed in the late 1980s, the WY transform [Bischof and Van Loan 1987] and the compact WY transform (CWY) [Schreiber and Van Loan 1989]. A third such transform was proposed and published by Walker in 1988 [Walker 1988] in the setting of a GMRES algorithm based on Householder transformations and rediscovered by Puglisi in 1992 in the setting of the QR factorization [Puglisi 1992][1]. Yet few in the numerical analysis community appear to be aware of these results as they relate to the CWY [Sun 1996]. It was a brief brainstorming session involving the authors of this article that independently rediscovered this result once again. We believe the result to be of sufficient importance that it warrants republishing.

In Section 2, we review the traditional way in which the CWY is computed. In Section 3, we present the main theorem that characterizes the accumulation of Householder transformations. In Section 4, we discuss opportunities that appear due to the alternative characterization. Remarks on how to modify LAPACK to accommodate the insights are given in Section 5. Experimental results are presented in Section 6, followed by concluding remarks in the final section.

2. COMPUTING THE COMPACT WY TRANSFORM

The following theorem presents the traditional formula for accumulating Householder transformations into a CWY:

THEOREM 1. *Let the matrix $U_{k-1} \in \mathbb{R}^{m\times k}$ have linearly independent columns. Partition U by columns as*

$$U_{k-1} = (u_0|u_1|\cdots|u_{k-1}),$$

and consider the vector $t = (\tau_0, \tau_1, \ldots, \tau_{k-1})^T$ with $\tau_i \neq 0$, $0 \leq i < k$. Then, there exists a unique nonsingular upper triangular matrix $S_{k-1} \in \mathbb{R}^{k\times k}$ such that

$$\left(I - \frac{u_0 u_0^T}{\tau_0}\right)\left(I - \frac{u_1 u_1^T}{\tau_1}\right)\cdots\left(I - \frac{u_{k-1} u_{k-1}^T}{\tau_{k-1}}\right) = \left(I - U_{k-1} S_{k-1} U_{k-1}^T\right).$$

The matrices $S_0, S_1, \ldots, S_{k-1}$ can be computed via the recurrence

$$S_0 = 1/\tau_0 \quad \textit{and} \quad S_i = \left(\begin{array}{c|c} S_{i-1} & -S_{i-1} U_{i-1}^T u_i/\tau_i \\ \hline 0 & 1/\tau_i \end{array}\right), \quad 1 \leq i < k. \tag{1}$$

PROOF. The recurrence gives the standard algorithm for computing the accumulation of Householder transformations into a CWY. It is proved by induction

[1]We emphasize that, while we will often refer to Puglisi's paper in this article, it is Walker who should be given credit for first proposing the methodology discussed in this article.

Partition $U \rightarrow \left(\begin{array}{c|c} U_L & U_R \end{array}\right)$, $t \rightarrow \left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right)$, $S \rightarrow \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline & S_{BR} \end{array}\right)$

where U_L has 0 columns, t_T has 0 rows, and S_{TL} is 0×0

while $m(S_{TL}) < m(S)$ **do**

Repartition

$$\left(\begin{array}{c|c} U_L & U_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_0 & u_1 & U_2 \end{array}\right),$$

$$\left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right) \rightarrow \left(\begin{array}{c} t_0 \\ \hline \tau_1 \\ \hline t_2 \end{array}\right), \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline & S_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} S_{00} & s_{01} & S_{02} \\ \hline & \sigma_{11} & s_{12}^T \\ \hline & & S_{22} \end{array}\right)$$

where u_1 has one column, and τ_1, σ_{11} are scalars

$s_{01} := U_0^T u_1$

$\sigma_{11} := 1/\tau_1 \; (= 2/u_1^T u_1)$

$s_{01} := -S_{00} s_{01} \sigma_{11}$

Continue with

$$\left(\begin{array}{c|c} U_L & U_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_0 & u_1 & U_2 \end{array}\right),$$

$$\left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right) \leftarrow \left(\begin{array}{c} t_0 \\ \hline \tau_1 \\ \hline t_2 \end{array}\right), \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline & S_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} S_{00} & s_{01} & S_{02} \\ \hline & \sigma_{11} & s_{12}^T \\ \hline & & S_{22} \end{array}\right)$$

endwhile

Fig. 1. Traditional algorithm for computing S.

on k. (In our theorem, we lift the restriction that each transformation must be a Householder transformation, a generalization that we will not use subsequently in the article.) □

An algorithm for computing this transformation based on (1) is given in Figure 1.

3. CENTRAL RESULT

We now state a theorem that will give a simpler characterization of the relation between U and S.

THEOREM 2. *Let $U \in \mathbb{R}^{m \times k}$ have linearly independent columns. Then, there exists a unique nonsingular upper triangular matrix $S \in \mathbb{R}^{k \times k}$ such that $I - USU^T$ is an orthogonal matrix. This matrix S satisfies $S = T^{-1}$ with $T + T^T = U^T U$, where $T \in \mathbb{R}^{k \times k}$ is itself a unique nonsingular upper triangular matrix.*

PROOF. We first prove existence. Consider U partitioned by columns as $U = (u_0 | \cdots | u_{k-1})$, and let $\tau_i = u_i^T u_i / 2$, $0 \leq i < k$. We then recognize $\tau_i \neq 0$, and each $(I - \frac{u_i u_i^T}{\tau_i})$ is a Householder transformation. Multiplying these Householder transformations together results in an orthogonal matrix. From this, Theorem 1 yields the desired nonsingular upper triangular matrix S.

Since $I - USU^T$ is orthogonal,

$$\begin{aligned} 0 &= I - (I - USU^T)(I - USU^T)^T = I - (I - USU^T)(I - US^T U^T) \\ &= I - (I - US^T U^T - USU^T + USU^T US^T U^T) \\ &= U[(S^T + S) - SU^T US^T] U^T. \end{aligned}$$

Thus, $S^T + S = SU^T US^T$ since U has full column rank. Now, as matrix S is required to be nonsingular, $S^{-1}(S^T + S)S^{-T} = S^{-1}SU^T US^T S^{-T}$, and therefore

$$S^{-1} + S^{-T} = U^T U. \tag{2}$$

Finally, replacing S^{-1} by T in (2), we find that $T = \text{striu}(U^T U) + \frac{1}{2}\text{diag}(U^T U)$ uniquely defines the upper triangular matrix T. Here striu(A) denotes the part of matrix A that lies strictly above the diagonal of that matrix, and diag(A) equals the diagonal matrix that has the same diagonal as A. □

Under the assumptions of this theorem, S can be computed by the following three steps:

(1) $S :=$ the upper triangular part of $U^T U$;
(2) Divide the diagonal elements of S by two;
(3) $S := S^{-1}$.

An algorithm for the first step is given in the top part of Figure 2, while an algorithm that combines the last two steps is given in the bottom part of that figure.

NOTE 1. *Puglisi arrived at the result in Theorem* 2 *by applying the Woodbury-Morrison formula to* $I - USU^T$. *We believe our proof is simpler and more revealing.*

The two algorithms in Figure 2 together implement *exactly* the same computation as the traditional algorithm in Figure 1 except that, rather than computing σ_{11} in three steps ($\sigma_{11} := u_1^T u_1$; $\sigma_{11} := \sigma_{11}/2$; $\sigma_{11} := 1/\sigma_{11}$), the traditional algorithm simply sets σ_{11} to τ_1, which has the same net result.

Update in Figure 1	Update in Figure 2
$s_{01} := U_0^T u_1$	$s_{01} := U_0^T u_1$
$\sigma_{11} := 1/\tau_1 \left(= 2/\left(u_1^T u_1\right)\right)$	$\begin{cases} \sigma_{11} := u_1^T u_1 \\ \sigma_{11} := \sigma_{11}/2 \\ \sigma_{11} := 1/\sigma_{11} \end{cases}$
$s_{01} := -S_{00} s_{01} \sigma_{11}$	$s_{01} := -S_{00} s_{01} \sigma_{11}$

Other than one additional recomputation of $u_1^T u_1/2$ per diagonal element of S, the two algorithms perform the same operations. Therefore, they will have very similar cost and numerical stability. This additional computation is an artifact of the fact that the level 3 Basic Linear Algebra Subprograms (BLAS) routine DSYRK [Dongarra et al. 1990], which would typically be used to compute $U^T U$, also recomputes the diagonal of the result. Clearly, σ_{11}, the diagonal element of S, could simply be set to $1/\tau_1$ in Figure 2. The computation of $U^T U$ and the inversion of S can be implemented using any algorithm for those operations, not just the ones in Figure 2.

NOTE 2. *Puglisi makes the same connection between the traditional algorithm for computing* S *and the separate steps just mentioned.*

Partition $U \rightarrow \left(\begin{array}{c|c} U_L & U_R \end{array} \right)$, $S \rightarrow \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline & S_{BR} \end{array} \right)$

where U_L has 0 columns and S_{TL} is 0×0

while $m(S_{TL}) < m(S)$ **do**

Repartition

$$\left(\begin{array}{c|c} U_L & U_R \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} U_0 & u_1 & U_2 \end{array} \right), \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline & S_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} S_{00} & s_{01} & S_{02} \\ \hline & \sigma_{11} & s_{12}^T \\ \hline & & S_{22} \end{array} \right)$$

where u_1 is a column and σ_{11} is a scalar

$s_{01} := U_0^T u_1$
$\sigma_{11} := u_1^T u_1$

Continue with

$$\left(\begin{array}{c|c} U_L & U_R \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} U_0 & u_1 & U_2 \end{array} \right), \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline & S_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} S_{00} & s_{01} & S_{02} \\ \hline & \sigma_{11} & s_{12}^T \\ \hline & & S_{22} \end{array} \right)$$

endwhile

Partition $S \rightarrow \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline & S_{BR} \end{array} \right)$

where S_{TL} is 0×0

while $m(S_{TL}) < m(S)$ **do**

Repartition

$$\left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline & S_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} S_{00} & s_{01} & S_{02} \\ \hline & \sigma_{11} & s_{12}^T \\ \hline & & S_{22} \end{array} \right)$$

where σ_{11} is a scalar

$\sigma_{11} := \sigma_{11}/2$
$\sigma_{11} := 1/\sigma_{11}$
$s_{01} := -S_{00} s_{01} \sigma_{11}$

Continue with

$$\left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline & S_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} S_{00} & s_{01} & S_{02} \\ \hline & \sigma_{11} & s_{12}^T \\ \hline & & S_{22} \end{array} \right)$$

endwhile

Fig. 2. Computing S as proposed in Section 3. Top: Compute $S := U^T U$ (upper triangular part only). Bottom: Divide the diagonal elements of S by 2 and compute $S := S^{-1}$.

4. OPPORTUNITIES

While the result in the previous section provides a simple theoretical characterization of the relation between the Householder vectors and the CWY, we now show how it provides opportunities for performance and numerical stability.

4.1 Potential Impact on Performance

The traditional algorithm in Figure 1 is rich in matrix-vector products, a level 2 BLAS [Dongarra et al. 1988] operation. By contrast, Steps (1)–(3) in Section 3 can inherently attain high performance: Step (1) can be implemented by a call to an optimized implementation of the level 3 BLAS

routine DSYRK, while the LAPACK routine DTRTRI can be used for Step (3). Typically, k is small enough so that the inversion of the $k \times k$ matrix in Step (3) will keep that matrix in cache memory, making that operation inherently efficient.

NOTE 3. *Puglisi makes the same observation.*

4.2 The UT Transform

$I - UT^{-1}U^T$ represents an alternative expression for the accumulation of the Householder transformations. This formulation eliminates the need for the k^3 floating-point operations (flops) required to compute $S := T^{-1}$. We call this formulation *the UT transform*.

The CWY is typically formed so that it can be applied to a matrix $A \in \mathbb{R}^{m \times n}$, as in the computation $A := (I - USU^T)A$. One can instead compute $(I - UT^{-1}U^T)A$. The parentheses in the following expressions indicate the order in which operations in these two approaches are typically performed:

$$A := A - U[S \underbrace{[U^T A]}_{W}] \quad \text{versus} \quad A := A - U[T^{-1} \underbrace{[U^T A]}_{W}].$$

The computation of SW and $T^{-1}W$, via the level 3 BLAS routines DTRMM and DTRSM, respectively, requires *exactly* the same number of flops. Thus, avoiding the inversion of matrix T translates directly into k^3 fewer flops being performed.

NOTE 4. *Puglisi makes the same observation.*

For different implementations of the BLAS, DTRSM may attain better or worse performance than DTRMM. This would influence whether to compute and use the UT transform or the CWY.

4.3 Potential Impact on Numerical Stability

Householder transformations are inherently used because of their exceptional stability properties. The CWY is known to inherit these properties. Nonetheless, it is also well known that computing $W := T^{-1}W$ as a triangular solve with multiple right-hand sides is numerically more stable than computing $W := SW$ after explicitly inverting $S := T^{-1}$. Thus, the UT transform is at least as stable as the CWY, and possibly more stable.

NOTE 5. *Puglisi makes a similar comment regarding stability.*

5. MODIFICATIONS TO LAPACK

We now give details of how minor modifications to LAPACK can be made to incorporate the insights in this article.

A detail that is not made obvious in the previous discussion is that the matrix U that stores the Householder vectors as they are computed during a QR factorization has the form $U = \binom{U_1}{U_2}$, where U_1 is unit lower triangular. Thus, the computation $S = U^T U$ can be broken down into $S := U_1^T U_1$, followed by

$S := \text{triu}(U_2^T U_2)$

Partition $U_1 \rightarrow \left(\begin{array}{c|c} U_L & U_R \end{array}\right)$, $t \rightarrow \left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right)$, $S \rightarrow \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline & S_{BR} \end{array}\right)$

where U_L has 0 columns, t_T has 0 rows, and S_{TL} is 0×0

while $m(S_{TL}) < m(S)$ **do**

Repartition

$\left(\begin{array}{c|c} U_L & U_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_0 & u_1 & U_2 \end{array}\right)$,

$\left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right) \rightarrow \left(\begin{array}{c} t_0 \\ \hline \tau_1 \\ \hline t_2 \end{array}\right)$, $\left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline & S_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} S_{00} & s_{01} & S_{02} \\ \hline & \sigma_{11} & s_{12}^T \\ \hline & & S_{22} \end{array}\right)$

where u_1 has one column, and τ_1, σ_{11} are scalars

Compute S**:**	**Compute** $T = S^{-1}$ **in** S**:**
$s_{01} := s_{01} + U_0^T u_1$	$s_{01} := s_{01} + U_0^T u_1$
$\sigma_{11} := 1/\tau_1$	$\sigma_{11} := \tau_1$
$s_{01} := -S_{00} s_{01} \sigma_{11}$	

Continue with

$\left(\begin{array}{c|c} U_L & U_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_0 & u_1 & U_2 \end{array}\right)$,

$\left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right) \leftarrow \left(\begin{array}{c} t_0 \\ \hline \tau_1 \\ \hline t_2 \end{array}\right)$, $\left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline & S_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} S_{00} & s_{01} & S_{02} \\ \hline & \sigma_{11} & s_{12}^T \\ \hline & & S_{22} \end{array}\right)$

endwhile

Fig. 3. Modification of traditional algorithm for computing S and T.

$S := S + U_2^T U_2$, computing only the upper triangular part. The term $U_2^T U_2$ is a simple call to DSYRK. The problem is that there is no routine in the BLAS or LAPACK that computes only the upper triangular part of $S = U_1^T U_1$, while taking advantage of the special structure of U_1.

To overcome this, let us examine routine DLARFT from LAPACK, which computes the matrix S via the algorithm in Figure 1. Now, S can be computed by initializing it to the upper triangular part of $U_2^T U_2$, changing the update $s_{01} := U_0^T u_1$ to $s_{01} := s_{01} + U_0^T u_1$ in Figure 1, and executing this modified algorithm with U_1 rather than all of U. Thus, first S is set to $U_2^T U_2$, after which the remaining computations are all accomplished by the modification given in Figure 3 (left). This approach casts most computations in terms of $U_2^T U_2$ (DSYRK) and, in one sweep, performs the remaining computation with matrices that are small enough to remain in cache. This is coded by modifying DLARFT, adding a call to DSYRK with U_2 before the loop, changing the upper limit of the loop from `N` (the row dimension of U) to `K` (the row dimension of U_1), and changing a `ZERO` to a `ONE` in the call to DGEMV so that the result of the matrix-vector multiply is added to s_{01}. Let us call the result DLARFT_NEW.

The new routine DLARFT_NEW can then be turned into a computation of T by further changing the algorithm in Figure 1, replacing $\sigma_{11} = 1/\tau_1$ by $\sigma_{11} = \tau_1$, and deleting the update $s_{01} = -S_{00} s_{01} \sigma_{11}$, as illustrated in Figure 3 (right). This translates to a change in one line of DLARFT_NEW and the deletion of one call to DTRMV. Applying the UT transform so computed requires only that a single call to DTRMM be changed to a call to DTRSM in DLARFB.

6. EXPERIMENTS

We demonstrate the potential of the alternative approaches by modifying the LAPACK routines for computing and applying the CWY, DLARFT and DLARFB, and measuring its effect on the LAPACK QR factorization routine, DGEQRF.

6.1 Implementation

Three different implementations were examined: **LAPACK**, the standard LAPACK implementation; **CWY-alt**, the modified LAPACK implementation based on the algorithm in Figure 3 (left); and **UT**, the modified LAPACK implementation based on the UT transform as described in Figure 3 (right).

6.2 Performance

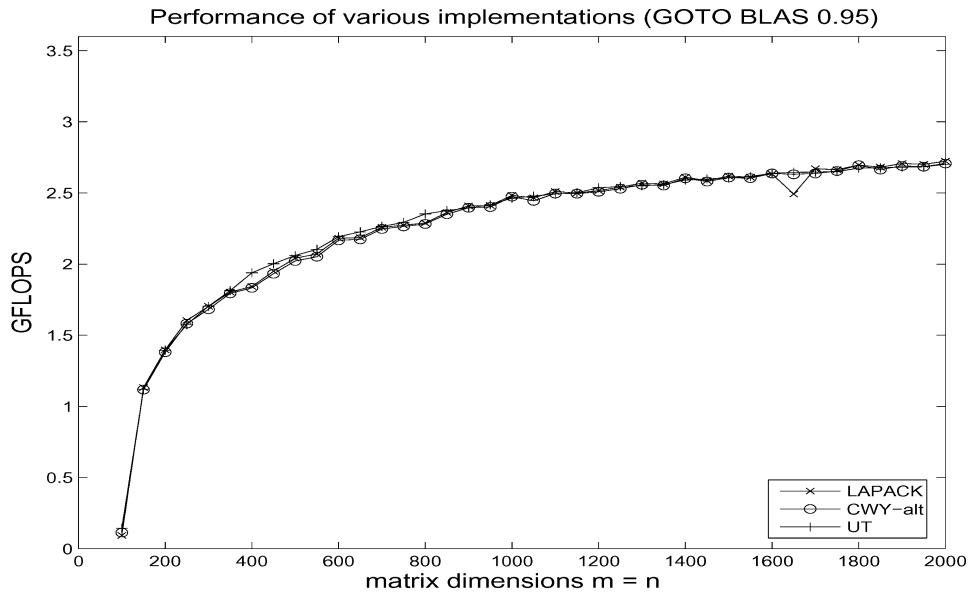
The impact of the described modifications was measured by computing the QR factorization of matrices of various sizes and using the result to solve a linear system (with a single right-hand side). The first target platform was an Intel Itanium 2 (900MHz) processor-based workstation, using the GOTO BLAS library, Release 0.95 [Goto 2005]. The results are reported in Figure 4. In all experiments, a blocksize of $k = 32$ was used.

Casting most computation in DLARFT in terms of DSYRK yields a slight degradation in performance. We speculate that is due to inefficiencies in the implementation of that routine for the specific matrix dimensions that are encountered in our computation. By switching to the implementation based on the UT transform, modest performance improvements are observed. As part of a QR factorization, the amount of computation that is performed in the routines that were optimized constitutes a lower order term so modest improvements are all that can be expected. The performance results in Figure 4 highlight the importance of how well the different kernels that are used by the algorithm are tuned.

NOTE 6. *Puglisi also comments on the inefficiency of the* DSYRK *operation in similar experiments.*

In Figure 5, we report performance attained on an eight CPU NEC SX-6 SMP server with a peak of 8 GFLOPS per CPU. Each CPU of this architecture is a vector processor, making it possible to highly and (more importantly) equally optimize any of the level 3 BLAS. While on a single CPU, no benefit is observed from computing the triangular matrix via SYRK, on multiple CPUs, a noticeable performance improvement results. This is because SYRK parallelizes better than the traditional algorithm for computing S which is rich in matrix-vector multiplication. Since the explicit inversion of the triangular matrix constitutes very little computation relative to the overall QR factorization, CWY-alt and UT attained essentially the same performance.

NOTE 7. *Walker originally proposed this methodology to improve parallelism and reduce communication on distributed memory architectures.*



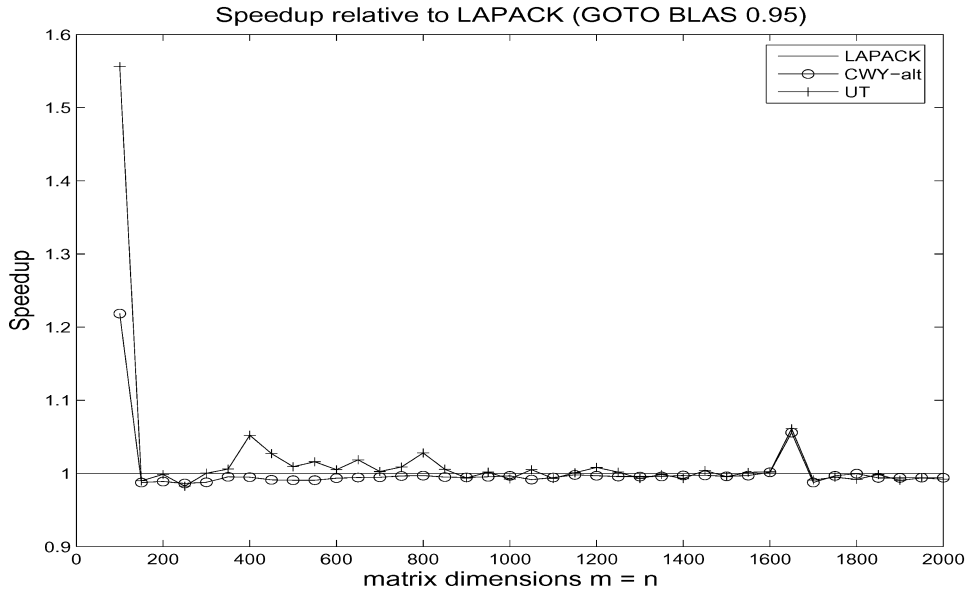


Fig. 4. Performance of the various implementations on an Intel Itanium 2 (900MHz) server (single CPU), linked to GOTO BLAS release 0.95.

6.3 Numerical Stability

The effects of the modifications on numerical stability were also experimentally examined and no meaningful improvements or degradations in the quality of the residual were observed.

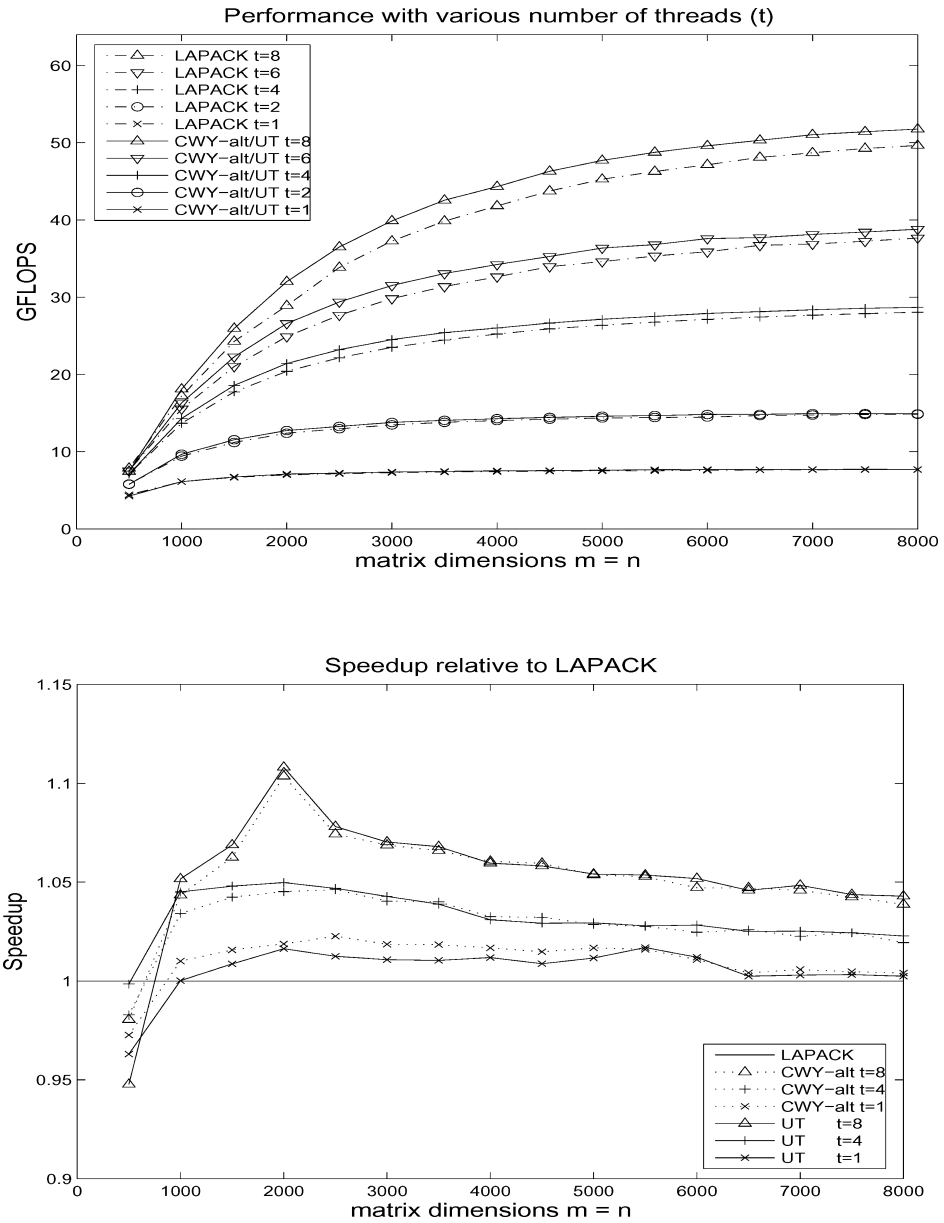


Fig. 5. Performance of LAPACK and UT on a NEC SX-6 SMP server. Note: the performance of CWY-alt and UT was virtually indistinguishable on this architecture.

7. CONCLUSION

In this article, an alternative characterization of the compact WY transform was given. The characterization suggests a simple approach to computing that transform and an alternative way of accumulating Householder transformations, the UT transform, which eliminates the cost of the inversion of a

triangular matrix. This alternative transform was already proposed, first by Walker and again by Puglisi, a result that appears to have been lost to the community. On sequential systems, the benefits of the methodology is highly dependent on the tuning of the BLAS library. Performance gains can be expected to be more significant on SMP and distributed memory architectures.

ACKNOWLEDGMENTS

We thank Thuan Cao for performing the experiments on the NEC SX-6 and Dr. Robert Schreiber for his encouragement.

REFERENCES

BISCHOF, C. AND VAN LOAN, C. 1987. The WY representation for products of Householder matrices. *SIAM J. Sci. Stat. Comput. 8*, 1 (Jan.), 2–13.

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

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

GOTO, K. 2005. `http://www.cs.utexas.edu/users/kgoto/`.

HOUSEHOLDER, A. S. 1958. Unitary triangularization of a nonsymmetric matrix. *J. ACM 5*, 339–342.

PUGLISI, C. 1992. Modification of the Householder method based on the compact wy representation. *SIAM J. Sci. Stat. Comput. 13*, 723–726.

SCHREIBER, R. AND VAN LOAN, C. 1989. A storage-efficient WY representation for products of Householder transformations. *SIAM J. Sci. Stat. Comput. 10*, 1 (Jan.), 53–57.

SUN, X. 1996. Aggregations of elementary transformations. Tech. Rep. Tech. rep. DUKE-TR-1996-03, Duke University.

WALKER, H. F. 1988. Implementation of the GMRES method using Householder transformations. *SIAM J. Sci. Stat. Comput. 9*, 1, 152–163.

Received August 2004; revised July 2005; accepted August 2005

Improving the Performance of Reduction to Hessenberg Form

GREGORIO QUINTANA-ORTÍ
Universidad Jaume I
and
ROBERT VAN DE GEIJN
The University of Texas at Austin

In this article, a modification of the blocked algorithm for reduction to Hessenberg form is presented that improves performance by shifting more computation from less efficient matrix-vector operations to highly efficient matrix-matrix operations. Significant performance improvements are reported relative to the performance achieved by the current LAPACK implementation.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra; G.4 [**Mathematical Software**]—*Efficiency*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Linear algebra, eigenvalue problems, reduction to condensed form

1. INTRODUCTION

The reduction to Hessenberg form is an important and time consuming step in the computation of the Schur decomposition of a square nonsymmetric matrix. The Schur decomposition itself is important since it solves the nonsymmetric eigenvalue problem [Golub and Van Loan 1996] and is a step towards the solution of the Sylvester equation and other linear algebra equations that arise in control theory [Sima 1996; Bartels and Stewart 1972; Golub et al. 1979; Golub and Van Loan 1996]. Thus, improving the performance of this computation impacts a number of important applications.

This research was partially sponsored by NSF grants ACI-0305163 and CCF-0342369 and an equipment donation from Hewlett-Packard.
Authors' addresses: G. Quintana-Ortí, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, Campus Riu Sec, 12.071—Castellón, Spain; email: gquintan@icc.uji.es; R. A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: rvdg@cs.utexas.edu.

Algorithms for the computation of the reduction to Hessenberg form date back to the early 1960s [Wilkinson 1965; Martin and Wilkinson 1968]. In the 1980s, it was observed that the key to attaining high performance on architectures with complex multilevel memories is to cast computation in terms of matrix-matrix operations, like the matrix-matrix product. These operations are supported by the level 3 Basic Linear Algebra Subprograms (BLAS) [Dongarra et al. 1990]. The idea is that such operations perform $O(n^3)$ floating point operations (flops) on $O(n^2)$ data, where n equals the matrix dimension of the operands. This allows data to be brought into fast cache memory, amortizing the cost of this data movement over a large number of computations. An algorithm that casts part of the computation required for the reduction in terms of matrix-matrix operations (level 3 BLAS) was first presented in Dongarra et al. [1989]. Currently the most widely used library for linear algebra operations, the Linear Algebra Package (LAPACK) [Anderson et al. 1999], incorporates one such algorithm in the routine DGEHRD.

As we will review later in this article, inherently a considerable part of the computation must be in terms of matrix-vector operations (level 2 BLAS [Dongarra et al. 1988]) that attain only a modest percentage of the peak performance of an architecture due to memory bandwidth limitations. Thus, even if the rest of the computation is cast in terms of level 3 BLAS, the part that requires level 2 BLAS will remain and will limit the percentage of peak that the algorithm can achieve. The problem with the implementation that is currently part of LAPACK is that it leaves more computation in terms of level 2 BLAS than necessary. It is this shortcoming that the algorithm in this article addresses.

The primary benefit of our new implementation is a shift of computation from level 2 to level 3 BLAS. A secondary benefit comes from the fact that the computation that is not in level 3 BLAS involves less data, improving data locality and reducing cache traffic during those operations: The current LAPACK implementation touches in every iteration (that is, n times) all columns to the right of the current column, while our algorithm only touches the part of those same columns that is below the current row.

The remainder of this article is organized as follows. Householder transformations and related transforms are reviewed in Section 2. The traditional (unblocked) algorithm for reduction to Hessenberg form is given in Section 3. Blocked algorithms are then described in Section 4. Performance results are given in Section 5, followed by concluding remarks in the final section.

2. HOUSEHOLDER TRANSFORMS

Given a nonzero vector $u \in \mathbb{R}^m$, a *Householder transformation* (or *reflector*) is defined by $H = I_m - uu^T/\tau$, where I_m denotes the (square) identity matrix of order m and $\tau = u^T u/2$ [Householder 1958]. It is an orthogonal matrix ($H^T H = HH^T = I_m$) and symmetric ($H^T = H$). This transformation has wide application in the solution of linear least-squares problems, the computation of orthonormal bases, and the solution of the algebraic eigenvalue problem.

Householder transformations will be used in this article to annihilate elements below the first subdiagonal of a given matrix. More precisely, given a

vector x, partition $x = \left(\begin{smallmatrix} \chi_1 \\ x_2 \end{smallmatrix}\right)$, where χ_1 equals the first element of x. Define the Householder vector associated with x as the vector $u = \left(\begin{smallmatrix} 1 \\ u_2 \end{smallmatrix}\right)$, where $u_2 = x_2/\rho_1$ with $\rho_1 = \chi_1 + \text{sign}(\chi_1)\|x\|_2$. Here sign$(\alpha)$ returns 1 or -1, depending on the sign of α. Let $\tau = u^T u/2$. Then $(I - uu^T/\tau)x = \{0 \backslash\backslash \beta\}$ where $\beta = -\text{sign}(\chi_1)\|x\|_2$, and $\{0 \backslash\backslash \beta\}$ equals the zero vector with the first element replaced by β. Let us introduce the notation $[\{u \backslash\backslash \beta\}, \tau] := \text{Hous}(x)$ for the operation that computes the aforementioned β, u, and τ from the given vector x. Here $\{u \backslash\backslash \beta\}$ indicates the vector u with the first element (which is implicitly equal to 1) overwritten by the value β.

Multiplying two or more Householder transformations results again in an orthogonal matrix, although not symmetric. This allows Householder transformations to be accumulated into a composed transformation. Traditionally, the *compact WY transform* (CWY) is used, which has the form $I - USU^T$, where the columns of U consist of the Householder vectors being accumulated. Let the columns of U_{k-1} equal the first k Householder vectors u_j, $0 \le j < k$,

$$\left(I - u_0 u_0^T/\tau_0\right) \cdots \left(I - u_{k-1} u_{k-1}^T/\tau_{k-1}\right) = \left(I - U_{k-1} S_{k-1} U_{k-1}^T\right),$$

where S_{k-1} is given by the induction

$$S_0 = 1/\tau_0 \quad \text{and} \quad S_k = \left(\begin{array}{c|c} S_{k-1} & -S_{k-1} U_{k-1} u_k/\tau_k \\ \hline 0 & 1/\tau_k \end{array}\right),$$

which is similar to the induction formula in Schreiber and Van Loan [1989].

A little-known alternative [Joffrain et al. ; Puglisi 1992; Walker 1988], which we call the UT transform, has the form $I - UT^{-1}U^T$ where

$$\left(I - u_0 u_0^T/\tau_0\right) \cdots \left(I - u_{k-1} u_{k-1}^T/\tau_{k-1}\right) = \left(I - U_{k-1} T_{k-1}^{-1} U_{k-1}^T\right),$$

with $T_0 = \tau_0$ and $T_k = \left(\begin{array}{c|c} T_{k-1} & U_{k-1} u_k \\ \hline 0 & \tau_k \end{array}\right)$. Note that T can be computed from U as $T = U^T U$ (upper triangular part only), followed by the dividing of the diagonal elements of the resulting T by two. Upper triangular matrices T and S are related by $S = T^{-1}$.

Both the CWY and the UT transform allow the application of a series of Householder transformations to be cast in terms of high-performance matrix-matrix operations:

$$(I - USU^T)A = A - U[\,S[\,\underbrace{U^T A}_{\text{GEMM}}\,]\,]}_{\text{TRMM}}}_{\text{GEMM}} \quad \text{and} \quad (I - UT^{-1}U^T)A = A - U[\,\underbrace{T^{-1}[\,\underbrace{U^T A}_{\text{GEMM}}\,]}_{\text{TRSM}}\,]}_{\text{GEMM}}.$$

3. NEW NOTATION FOR THE TRADITIONAL REDUCTION ALGORITHM

In this section, we discuss the basic idea behind the algorithm for computing the Hessenberg reduction of a square matrix A: $A = QBQ^T$, where Q is unitary and B is upperHessenberg (B has zeroes below the first subdiagonal). We will see that Q will be computed as a sequence of Householder transformations,

that B can overwrite A, and that the Householder vectors can overwrite the parts of A below the first subdiagonal.

Let us denote the original contents of matrix A as $\hat{A}$ and assume that A is to be overwritten by the upperHessenberg matrix. Partition

$$A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) \quad \text{and} \quad \hat{A} \rightarrow \left(\begin{array}{c|c} \hat{\alpha}_{11} & \hat{a}_{12}^T \\ \hline \hat{a}_{21} & \hat{A}_{22} \end{array}\right).$$

Let $[\{u\backslash\backslash\beta\}_{21}, \tau_1] = \text{Hous}(a_{21})$, and $H_0 = H(u_{21}, \tau_1) \equiv I - u_{21}u_{21}^T/\tau_1$ so that $\{0\backslash\backslash\beta\}_{21} = H_0 a_{21}$. Then the first step of the reduction updates

$$A := \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) := \left(\begin{array}{c|c} 1 & 0 \\ \hline 0 & H_0 \end{array}\right) \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) \left(\begin{array}{c|c} 1 & 0 \\ \hline 0 & H_0 \end{array}\right) = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T H_0 \\ \hline \{0\backslash\backslash\beta\}_{21} & H_0 A_{22} H_0 \end{array}\right).$$

To complete an upperHessenberg reduction, the procedure continues by computing Householder transformations from the updated A_{22} and applying them to that matrix as well as the part of matrix A that appears above A_{22}.

This procedure can be described more completely as follows. After k steps, partition A and $\hat{A}$ as

$$A \rightarrow \left(\begin{array}{c|c} B_{TL} & A_{TR} \\ \hline \{0\backslash\backslash\beta\}_{BL} & A_{BR} \end{array}\right) \quad \text{and} \quad \hat{A} \rightarrow \left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right),$$

where B_{TL} and $\hat{A}_{TL}$ are $k \times k$, B_{TL} is upperHessenberg, and $\{0\backslash\backslash\beta\}_{BL}$ is the matrix of all zeroes except with β_{BL} in the top-right corner. Notice that by now A has been computed from $\hat{A}$ by computing $\{\check{H}_0, \ldots, \check{H}_{k-1}\}$ so that

$$A = \check{H}_{k-1} \cdots \check{H}_0 \hat{A} \check{H}_0 \cdots \check{H}_{k-1}, \quad \text{where } \check{H}_j = \left(\begin{array}{c|c} I_{j+1} & 0 \\ \hline 0 & H_j \end{array}\right).$$

Let us examine how A must be updated as part of the kth step. Repartition

$$\left(\begin{array}{c|c} B_{TL} & A_{TR} \\ \hline \{0\backslash\backslash\beta\}_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} B_{00} & a_{01} & A_{02} \\ \hline \{0\backslash\backslash\beta\}_{10}^T & \alpha_{11} & a_{12}^T \\ \hline 0 & a_{21} & A_{22} \end{array}\right),$$

where $\{0\backslash\backslash\beta\}_{10}^T$ equals the row vector of all zeroes except for the last element, which equals β_{10}. H_k is now computed as $H_k = H(u_{21}, \tau_1)$ where $[\{u\backslash\backslash\beta\}_{21}, \tau_1] = \text{Hous}(a_{21})$ and A is updated with

$$\left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & H_k \end{array}\right) \left(\begin{array}{c|c|c} B_{00} & a_{01} & A_{02} \\ \hline \{0\backslash\backslash\beta\}_{10}^T & \alpha_{11} & a_{12}^T \\ \hline 0 & a_{21} & A_{22} \end{array}\right) \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & H_k \end{array}\right)$$

$$= \left(\begin{array}{c|c|c} B_{00} & a_{01} & A_{02} H_k \\ \hline \{0\backslash\backslash\beta\}_{10}^T & \alpha_{11} & a_{12}^T H_k \\ \hline 0 & \{0\backslash\backslash\beta\}_{21} & H_k A_{22} H_k \end{array}\right).$$

Algorithm: $[A, t] := \text{HESREDUNB}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} \{U\backslash\backslash B\}_{TL} & A_{TR} \\ \hline \{U\backslash\backslash\beta\}_{BL} & A_{BR} \end{array}\right)$, $t \rightarrow \left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right)$

where $\{U\backslash\backslash B\}_{TL}$ is 0×0 and t_T has 0 elements

while $m(A_{TL}) < m(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} \{U\backslash\backslash B\}_{TL} & A_{TR} \\ \hline \{U\backslash\backslash\beta\}_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} \{U\backslash\backslash B\}_{00} & a_{01} & A_{02} \\ \hline \{u\backslash\backslash\beta\}_{10}^T & \alpha_{11} & a_{12}^T \\ \hline U_{20} & a_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right) \rightarrow \left(\begin{array}{c} t_0 \\ \hline \tau_1 \\ \hline t_2 \end{array}\right)$$

where α_{11} and τ_1 are scalars

$[\{u\backslash\backslash\beta\}_{21}, \tau_1] := \text{Hous}(a_{21})$ ($\{u\backslash\backslash\beta\}_{21}$ overwrites a_{21})

$A_{02} := A_{02}H(u_{21}, \tau_1)$

$a_{12}^T := a_{12}^T H(u_{21}, \tau_1)$

$A_{22} := H(u_{21}, \tau_1)A_{22}H(u_{21}, \tau_1)$

} via the steps in (1)–(4).

Continue with

$$\left(\begin{array}{c|c} \{U\backslash\backslash B\}_{TL} & A_{TR} \\ \hline \{U\backslash\backslash\beta\}_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} \{U\backslash\backslash B\}_{00} & b_{01} & A_{02} \\ \hline \{u\backslash\backslash\beta\}_{10}^T & \beta_{11} & a_{12}^T \\ \hline U_{20} & \{u\backslash\backslash\beta\}_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right) \leftarrow \left(\begin{array}{c} t_0 \\ \hline \tau_1 \\ \hline t_2 \end{array}\right)$$

endwhile

Fig. 1. Unblocked reduction to upperHessenberg form. In our algorithms "Repartition ..." and "Continue with ..." are used to indicate progress through the matrix.

The thick lines that indicate progress through the matrix can then be moved to include the next diagonal element:

$$\left(\begin{array}{c|c} B_{TL} & A_{TR} \\ \hline \{0\backslash\backslash\beta\}_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} B_{00} & a_{01} & A_{02}H_k \\ \hline \{0\backslash\backslash\beta\}_{10}^T & \alpha_{11} & a_{12}^T H_k \\ \hline 0 & \{0\backslash\backslash\beta\}_{21} & H_k A_{22} H_k \end{array}\right).$$

We finish this section by noting that vector u_k can be stored in the elements of A from which it was computed since it, by design, has a first element that equals 1, which, therefore, need not be stored. The scalars τ_k are typically stored in an auxiliary vector. Thus, after k steps A contains

$$\left(\begin{array}{c|c} \{U\backslash\backslash B\}_{TL} & A_{TR} \\ \hline \{U\backslash\backslash\beta\}_{BL} & A_{BR} \end{array}\right),$$

which is meant to indicate that the upperHessenberg matrix B_{TL} is stored in the upperHessenberg part of $\{U\backslash\backslash B\}_{TL}$, the element β_{BL} is stored in the top-right element of $\{U\backslash\backslash\beta\}_{BL}$, and the kth column[1] of A stores u_k below the first subdiagonal of that column. The complete algorithm is now given in Figure 1.

In Figure 1, it is important to realize that $H(u_{21}, \tau_1)$ is never explicitly formed, and the following formulas:

$$\begin{aligned} A_{02} &:= A_{02}H(u_{21}, \tau_1) = A_{02}\big(I - u_{21}u_{21}^T/\tau_1\big), \\ a_{12}^T &:= a_{12}^T H(u_{21}, \tau_1) = a_{12}^T\big(I - u_{21}u_{21}^T/\tau_1\big), \\ A_{22} &:= H(u_{21}, \tau_1)A_{22}H(u_{21}, \tau_1) = \big(I - u_{21}u_{21}^T/\tau_1\big)A_{22}\big(I - u_{21}u_{21}^T/\tau_1\big) \end{aligned}$$

[1]The 0th column of A is the left-most column here since we start indexing at 0.

can be implemented as

$$\left(\begin{array}{c} v_{01} \\ \hline \nu_{11} \\ \hline v_{21} \end{array}\right) := \left(\begin{array}{c} A_{02} \\ \hline a_{12}^T \\ \hline A_{22} \end{array}\right) u_{21}, \tag{1}$$

$$\left(\begin{array}{c} A_{02} \\ \hline a_{12}^T \\ \hline A_{22} \end{array}\right) := \left(\begin{array}{c} A_{02} \\ \hline a_{12}^T \\ \hline A_{22} \end{array}\right) - \left(\begin{array}{c} v_{01} \\ \hline \nu_{11} \\ \hline v_{21} \end{array}\right) u_{21}^T/\tau_1, \tag{2}$$

$$w_{21}^T := u_{21}^T A_{22}, \tag{3}$$

$$A_{22} := A_{22} - u_{21} w_{21}^T/\tau_1. \tag{4}$$

For the step where $\{U \backslash\backslash B\}_{TL}$ is $k \times k$, the cost of each computation in (1) and (2) is roughly $2n(n-k-1)$ flops, while each cost in (3) and (4) is roughly $2(n-k-1)^2$ flops. The total cost for reducing $A \in \mathbb{R}^{n\times n}$ is thus approximately

$$\sum_{k=0}^{n-1} (4n(n-k-1) + 4(n-k-1)^2) \text{ flops} \approx \frac{10}{3} n^3 \text{ flops}.$$

4. BLOCKED ALGORITHMS

In Section 2, we noted that, by accumulating multiple Householder transformations into a single transform, higher performance can be achieved when these transformations are to be applied to a matrix. The complication is that A must be updated with part of the computations in (1)–(4) before the next Householder transform can be computed. We first show how to progress to the point where a number of Householder transformations have been accumulated, after which we show how to then cast the remainder of the computation mostly in terms of matrix-matrix products.

4.1 Building Up a Block

We will need a temporary matrix $V \in \mathbb{R}^{n\times b}$ in which to store the vectors v that appeared in (1), and a matrix $T \in \mathbb{R}^{b\times b}$ that appears in the UT transform. In this discussion, we will also treat $U \in \mathbb{R}^{n\times b}$, which stored the Householder vectors, as a separate matrix although in practice it overwrites part of A.

Partition all matrices involved as

$$A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \quad \hat{A} = \left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right), \quad U = \left(\begin{array}{c|c} U_{TL} & 0 \\ \hline U_{BL} & U_{BR} \end{array}\right),$$

$$V = \left(\begin{array}{c|c} V_{TL} & V_{TR} \\ \hline V_{BL} & V_{BR} \end{array}\right), \quad \text{and} \quad T = \left(\begin{array}{c|c} T_{TL} & T_{TR} \\ \hline 0 & T_{BR} \end{array}\right),$$

where $X_{TL} \in \mathbb{R}^{k\times k}$, $k < b$, for $X \in \{A, \hat{A}, U, V, T\}$. Consider

$$\begin{aligned} &\left(I - u_{k-1}u_{k-1}^T/\tau_{k-1}\right) \cdots \left(I - u_0 u_0^T/\tau_0\right) \hat{A} \left(I - u_0 u_0^T/\tau_0\right) \cdots \left(I - u_{k-1}u_{k-1}^T/\tau_{k-1}\right) \\ &= \left(I - \left(\frac{U_{TL}}{U_{BL}}\right) T_{TL}^{-T} \left(\frac{U_{TL}}{U_{BL}}\right)^T\right) \left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right) \left(I - \left(\frac{U_{TL}}{U_{BL}}\right) T_{TL}^{-1} \left(\frac{U_{TL}}{U_{BL}}\right)^T\right) \end{aligned}$$

$$= \left(I - \left(\frac{U_{TL}}{U_{BL}} \right) T_{TL}^{-T} \left(\frac{U_{TL}}{U_{BL}} \right)^T \right) \left(\left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array} \right) - \left(\frac{V_{TL}}{V_{BL}} \right) T_{TL}^{-1} \left(\frac{U_{TL}}{U_{BL}} \right)^T \right),$$

where $\left(\frac{V_{TL}}{V_{BL}} \right) = \left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array} \right) \left(\frac{U_{TL}}{U_{BL}} \right)$. The idea is that not all of this has overwritten A. Only the first k columns have been updated with that part of the desired Hessenberg matrix:

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right) \text{ currently contains } \left(\begin{array}{c|c} B_{TL} & \hat{A}_{TR} \\ \hline \{0 \backslash\backslash \beta\}_{BL} & \hat{A}_{BR} \end{array} \right),$$

where

$$\left(\frac{B_{TL}}{\{0 \backslash\backslash \beta\}_{BL}} \right) = \left(I - \left(\frac{U_{TL}}{U_{BL}} \right) T_{TL}^{-T} \left(\frac{U_{TL}}{U_{BL}} \right)^T \right) \left(\left(\frac{\hat{A}_{TL}}{\hat{A}_{BL}} \right) - \left(\frac{V_{TL}}{V_{BL}} \right) T_{TL}^{-1} U_{TL}^T \right).$$

The question now becomes how to update the next column of A so that the next Householder transform can be computed (the next column of U), from which then the next columns of V and T can then be computed. Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array} \right), \quad \left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} \hat{A}_{00} & \hat{a}_{01} & \hat{A}_{02} \\ \hline \hat{a}_{10}^T & \hat{\alpha}_{11} & \hat{a}_{12}^T \\ \hline \hat{A}_{20} & \hat{a}_{21} & \hat{A}_{22} \end{array} \right),$$

$$\left(\begin{array}{c|c} V_{TL} & V_{TR} \\ \hline V_{BL} & V_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} V_{00} & v_{01} & V_{02} \\ \hline v_{10}^T & \nu_{11} & v_{12}^T \\ \hline V_{20} & v_{21} & V_{22} \end{array} \right), \quad \left(\begin{array}{c|c} U_{TL} & 0 \\ \hline U_{BL} & U_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & 0 & 0 \\ \hline u_{10}^T & 0 & 0 \\ \hline U_{20} & u_{21} & U_{22} \end{array} \right),$$

and

$$\left(\begin{array}{c|c} T_{TL} & T_{TR} \\ \hline 0_{BL} & T_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} T_{00} & t_{01} & T_{02} \\ \hline 0 & \tau_{11} & t_{12}^T \\ \hline 0 & 0 & T_{22} \end{array} \right).$$

The next column of A must be updated by

$$\left(\frac{b_{01}}{\beta_{11}} \right) := \left(I - \left(\frac{U_{00}}{u_{10}^T} \right) T_{00}^{-T} \left(\frac{U_{00}}{u_{10}^T} \right)^T \right) \left(\left(\frac{\hat{a}_{01}}{\hat{\alpha}_{11}} \right) - \left(\frac{V_{00}}{v_{10}^T} \right) T_{00}^{-1} u_{10} \right)$$

before the next Householder vector can be computed from the so updated b_{21}. After this, the next columns of V and T can be computed by the formulas

$$\left(\frac{v_{01}}{\nu_{11}} \right) := \left(\begin{array}{c|c|c} \hat{A}_{00} & \hat{a}_{01} & \hat{A}_{02} \\ \hline \hat{a}_{10}^T & \hat{\alpha}_{11} & \hat{a}_{12}^T \\ \hline \hat{A}_{20} & \hat{a}_{21} & \hat{A}_{22} \end{array} \right) \left(\frac{0}{u_{21}} \right) = \left(\frac{\hat{A}_{02} u_{21}}{\hat{a}_{12}^T u_{21}} \right) \tag{5}$$

Algorithm: $[A, V, T] := \text{HesRedBuildBlk}(A, V, T)$

Partition $A \rightarrow \left(\begin{array}{c|c} \{U\backslash\backslash B\}_{TL} & A_{TR} \\ \hline \{U\backslash\backslash\beta\}_{BL} & A_{BR} \end{array}\right)$, $V \rightarrow \left(\begin{array}{c|c} V_{TL} & V_{TR} \\ \hline V_{BL} & V_{BR} \end{array}\right)$, $T \rightarrow \left(\begin{array}{c|c} T_{TL} & T_{TR} \\ \hline 0 & T_{BR} \end{array}\right)$

where $\{U\backslash\backslash B\}_{TL}$ is 0×0, V_{TL} is 0×0, T_{TL} is 0×0

while $n(V_{TL}) < n(V)$ **do**

Repartition

$$\left(\begin{array}{c|c} \{U\backslash\backslash B\}_{TL} & A_{TR} \\ \hline \{U\backslash\backslash\beta\}_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} \{U\backslash\backslash B\}_{00} & a_{01} & A_{02} \\ \hline \{u\backslash\backslash\beta\}_{10}^T & \alpha_{11} & a_{12}^T \\ \hline U_{20} & a_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c|c} V_{TL} & V_{TR} \\ \hline V_{BL} & V_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} V_{00} & v_{01} & V_{02} \\ \hline v_{10}^T & \nu_{11} & v_{12}^T \\ \hline V_{20} & v_{21} & V_{22} \end{array}\right), \left(\begin{array}{c|c} T_{TL} & T_{TR} \\ \hline 0 & T_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} T_{00} & t_{01} & T_{02} \\ \hline 0 & \tau_{11} & t_{12}^T \\ \hline 0 & 0 & T_{22} \end{array}\right)$$

where α_{11} is 1×1, ν_{11} is 1×1, τ_{11} is 1×1

$$\left(\begin{array}{c} b_{01} \\ \hline \beta_{11} \\ \hline b_{21} \end{array}\right) := \left(I - \left(\begin{array}{c} U_{00} \\ \hline u_{10}^T \\ \hline U_{20} \end{array}\right) T_{00}^{-T} \left(\begin{array}{c} U_{00} \\ \hline u_{10}^T \\ \hline U_{20} \end{array}\right)^T\right) \left(\left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right) - \left(\begin{array}{c} V_{00} \\ \hline v_{10}^T \\ \hline V_{20} \end{array}\right) T_{00}^{-1} u_{10}\right)$$

via the steps

$$\begin{cases} y_{10} := T_{00}^{-1} u_{10} \\ \left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right) := \left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right) - \left(\begin{array}{c} V_{00} \\ \hline v_{10}^T \\ \hline V_{20} \end{array}\right) y_{10} \\ \left(\begin{array}{c} b_{01} \\ \hline \beta_{11} \\ \hline b_{21} \end{array}\right) := \left(I - \left(\begin{array}{c} U_{00} \\ \hline u_{10}^T \\ \hline U_{20} \end{array}\right) T_{00}^{-T} \left(\begin{array}{c} U_{00} \\ \hline u_{10}^T \\ \hline U_{20} \end{array}\right)^T\right) \left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right) \end{cases}$$

$[\{u\backslash\backslash\beta\}_{21}, \tau_{11}] := \text{Hous}(a_{21})$ ($\{u\backslash\backslash\beta\}_{21}$ overwrites a_{21})

$$\left(\begin{array}{c} v_{01} \\ \hline \nu_{11} \\ \hline v_{21} \end{array}\right) := \left(\begin{array}{c} \hat{A}_{02} u_{21} \\ \hline \hat{a}_{12}^T u_{21} \\ \hline \hat{A}_{22} u_{21} \end{array}\right); \quad \left(\begin{array}{c} t_{01} \\ \hline \tau_{11} \\ \hline 0 \end{array}\right) := \left(\begin{array}{c} U_{20}^T u_{21} \\ \hline \tau_{11} \\ \hline 0 \end{array}\right)$$

Continue with

$$\left(\begin{array}{c|c} \{U\backslash\backslash B\}_{TL} & A_{TR} \\ \hline \{U\backslash\backslash\beta\}_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} \{U\backslash\backslash B\}_{00} & b_{01} & A_{02} \\ \hline \{u\backslash\backslash\beta\}_{10}^T & \beta_{11} & a_{12}^T \\ \hline U_{20} & \{u\backslash\backslash\beta\}_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c|c} V_{TL} & V_{TR} \\ \hline V_{BL} & V_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} V_{00} & v_{01} & V_{02} \\ \hline v_{10}^T & \nu_{11} & v_{12}^T \\ \hline V_{20} & v_{21} & V_{22} \end{array}\right), \left(\begin{array}{c|c} T_{TL} & T_{TR} \\ \hline 0 & T_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} T_{00} & t_{01} & T_{02} \\ \hline 0 & \tau_{11} & t_{12}^T \\ \hline 0 & 0 & T_{22} \end{array}\right)$$

endwhile

Fig. 2. Algorithm for building up blocks for the blocked algorithm in Figure 3.

and

$$\left(\begin{array}{c} t_{01} \\ \hline \tau_{11} \\ \hline 0 \end{array}\right) := \left(\begin{array}{c} U_{20}^T u_{21} \\ \hline \tau_{11} \\ \hline 0 \end{array}\right),$$

where τ_{11} is the value returned by the routine that computes the Householder reflection. We note that V here is accumulated since, at some future point, this next column of V will be needed in order to update the next column of A.

An algorithm that embodies the above insights is given in Figure 2.

4.2 A New Blocked Algorithm

Now we show how a blocked algorithm can be achieved by repeatedly performing the steps in Section 4.1.

Once again, assume the computation has proceeded to where

$$A \rightarrow \left(\begin{array}{c|c} \{U\backslash\backslash B\}_{TL} & A_{TR} \\ \hline \{U\backslash\backslash \beta\}_{BL} & A_{BR} \end{array}\right),$$

where $B_{TL} \in \mathbb{R}^{k\times k}$ and A_{TR} and A_{BR} have been updated according to the Householder transformations computed so far: $A = \check{H}_{k-1}\cdots\check{H}_0\hat{A}\check{H}_0\cdots\check{H}_{k-1}$. Repartition

$$\left(\begin{array}{c|c} \{U\backslash\backslash B\}_{TL} & A_{TR} \\ \hline \{U\backslash\backslash \beta\}_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} \{U\backslash\backslash B\}_{00} & A_{01} & A_{02} \\ \hline \{U\backslash\backslash \beta\}_{10} & A_{11} & A_{12} \\ \hline U_{20} & A_{21} & A_{22} \end{array}\right),$$

where $A_{11} \in \mathbb{R}^{b\times b}$. The idea now is to call the algorithm in Figure 2 to compute

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) := \left(\begin{array}{c|c} \{U\backslash\backslash B\}_{11} & A_{12} \\ \hline \{U\backslash\backslash \beta\}_{21} & A_{22} \end{array}\right), \quad \left(\begin{array}{c} V_1 \\ \hline V_2 \end{array}\right), \quad \text{and} \quad T_1,$$

where A_{12} and A_{22} are not updated yet. Upon return, the following computations still need to be performed on the nonblank submatrices:

$$\left(\begin{array}{c|c|c} & A_{01} & A_{02} \\ \hline & & A_{12} \\ \hline & & A_{22} \end{array}\right).$$

These parts of the matrix must then be updated by

$$\begin{aligned} (A_{01} \mid A_{02}) &:= (A_{01} \mid A_{02})\left(I - \left(\frac{U_{11}}{U_{21}}\right) T_1^{-1} \left(\frac{U_{11}}{U_{21}}\right)^T\right) \\ &= (A_{01} \mid A_{02}) - V_0 T_1^{-1} \left(\frac{U_{11}}{U_{21}}\right)^T \end{aligned}$$

and

$$\left(\frac{A_{12}}{A_{22}}\right) := \left(I - \left(\frac{U_{11}}{U_{21}}\right) T_1^{-T} \left(\frac{U_{11}}{U_{21}}\right)^T\right)\left(\left(\frac{A_{12}}{A_{22}}\right) - \left(\frac{V_1}{V_2}\right) T_1^{-1} U_{21}^T\right).$$

A complete blocked algorithm based on these insights given in Figure 3.

4.3 Outline of LAPACK-Style Algorithm

The implementation that is currently part of LAPACK updates A slightly differently. Consider the repartitioning

$$\left(\begin{array}{c|c} \{U\backslash\backslash B\}_{TL} & A_{TR} \\ \hline \{U\backslash\backslash \beta\}_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} \{U\backslash\backslash B\}_{00} & A_{01} & A_{02} \\ \hline \{U\backslash\backslash \beta\}_{10} & A_{11} & A_{12} \\ \hline U_{20} & A_{21} & A_{22} \end{array}\right),$$

Algorithm: $[A, V, T] := \text{HESREDBLK}(A, V, T)$

Partition $A \rightarrow \left(\begin{array}{c|c} \{U\backslash\backslash B\}_{TL} & A_{TR} \\ \hline \{U\backslash\backslash\beta\}_{BL} & A_{BR} \end{array}\right)$, $V \rightarrow \left(\begin{array}{c} V_T \\ \hline V_B \end{array}\right)$, $T \rightarrow \left(\begin{array}{c} T_T \\ \hline T_B \end{array}\right)$

where $\{U\backslash\backslash B\}_{TL}$ is 0×0, V_T has 0 rows, T_T has 0 rows

while $m(A_{TL}) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} \{U\backslash\backslash B\}_{TL} & A_{TR} \\ \hline \{U\backslash\backslash\beta\}_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} \{U\backslash\backslash B\}_{00} & A_{01} & A_{02} \\ \hline \{U\backslash\backslash\beta\}_{10} & A_{11} & A_{12} \\ \hline U_{20} & A_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c} V_T \\ \hline V_B \end{array}\right) \rightarrow \left(\begin{array}{c} V_0 \\ \hline V_1 \\ \hline V_2 \end{array}\right), \left(\begin{array}{c} T_T \\ \hline T_B \end{array}\right) \rightarrow \left(\begin{array}{c} T_0 \\ \hline T_1 \\ \hline T_2 \end{array}\right)$$

where A_{11} is $b \times b$, V_1 has b rows, T_1 has b rows

$$\left[\left(\begin{array}{c|c} \{U\backslash\backslash B\}_{11} & A_{12} \\ \hline \{U\backslash\backslash\beta\}_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} V_1 \\ \hline V_2 \end{array}\right), T_1\right] := \text{HESREDBUILDBLK}\left(\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} V_1 \\ \hline V_2 \end{array}\right), T_1\right)$$

$$V_0 := \left(\begin{array}{c|c} A_{01} & A_{02} \end{array}\right) \left(\begin{array}{c} U_{11} \\ \hline U_{12} \end{array}\right)$$

$$\left(\begin{array}{c|c} A_{01} & A_{02} \end{array}\right) := \left(\begin{array}{c|c} B_{01} & A_{02} \end{array}\right) = \left(\begin{array}{c|c} A_{01} & A_{02} \end{array}\right) - V_0 T_1^{-1} \left(\begin{array}{c} U_{11} \\ \hline U_{21} \end{array}\right)^T$$

$$\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) := \left(I - \left(\begin{array}{c} U_{11} \\ \hline U_{21} \end{array}\right) T_1^{-T} \left(\begin{array}{c} U_{11} \\ \hline U_{21} \end{array}\right)^T\right) \left(\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) - \left(\begin{array}{c} V_1 \\ \hline V_2 \end{array}\right) T_1^{-1} U_{21}^T\right).$$

Continue with

$$\left(\begin{array}{c|c} \{U\backslash\backslash B\}_{TL} & A_{TR} \\ \hline \{U\backslash\backslash\beta\}_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} \{U\backslash\backslash B\}_{00} & B_{01} & A_{02} \\ \hline \{U\backslash\backslash\beta\}_{10} & \{U\backslash\backslash B\}_{11} & A_{12} \\ \hline U_{20} & \{U\backslash\backslash\beta\}_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c} V_T \\ \hline V_B \end{array}\right) \leftarrow \left(\begin{array}{c} V_0 \\ \hline V_1 \\ \hline V_2 \end{array}\right), \left(\begin{array}{c} T_T \\ \hline T_B \end{array}\right) \leftarrow \left(\begin{array}{c} T_0 \\ \hline T_1 \\ \hline T_2 \end{array}\right)$$

endwhile

Fig. 3. Blocked algorithm for computing the reduction to upperHessenberg form.

where $A_{11} \in \mathbb{R}^{b \times b}$. In the LAPACK implementation, the routine equivalent to HESSREDBUILDBLK, DLAHRD, returns having computed V_0, V_1, and V_2, and having updated A_{01} as well as A_{11} and A_{21} with the final results for those submatrices. Upon return, the updates still need to be performed on the nonblank submatrices:

$$\left(\begin{array}{c|c|c} & & A_{02} \\ \hline & & A_{12} \\ \hline & & A_{22} \end{array}\right).$$

These parts of the matrix must then be updated by

$$A_{02} := A_{02} - V_0 T_1^{-1} U_{21}^T$$

and

$$\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) := \left(I - \left(\begin{array}{c} U_{11} \\ \hline U_{21} \end{array}\right) T_1^{-T} \left(\begin{array}{c} U_{11} \\ \hline U_{21} \end{array}\right)^T\right) \left(\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) - \left(\begin{array}{c} V_1 \\ \hline V_2 \end{array}\right) T_1^{-1} U_{21}^T\right).$$

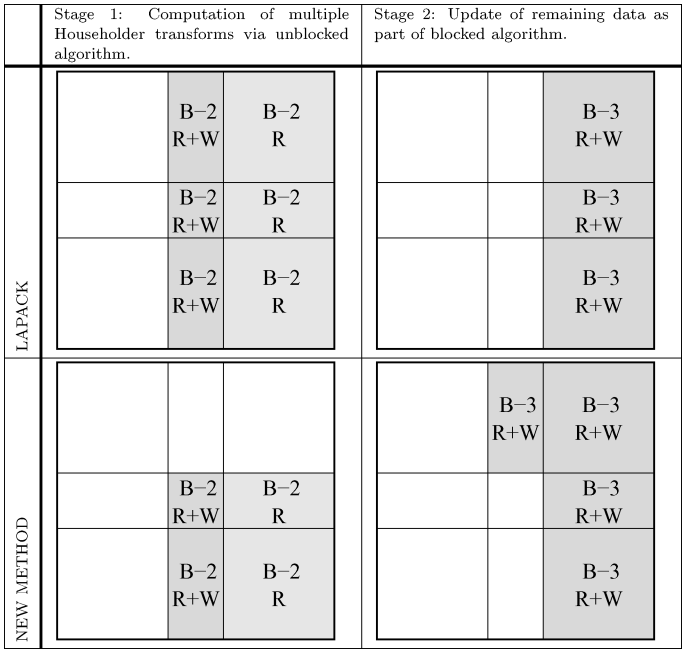


Fig. 4. Areas touched (read/written) in every stage of both algorithms. B-2 stands for BLAS-2 computations, B-3 stands for BLAS-3 computations, R stands for reading operation, W stands for writing operation. White areas are untouched (neither read nor written).

An inefficiency lies in the fact that the update of A_{01} as well as the computation of V_0 are cast in terms of level 2 BLAS rather than level 3 BLAS.

An additional performance hit comes from the fact that the LAPACK implementation touches in every iteration (that is, n times) the entire rest of the matrix (A_{TR}, and A_{BR}). By contrast, in every iteration, the algorithm in Section 4.2 only touches A_{BR}. This improves data locality and reduces cache traffic.

We note that the LAPACK-style algorithm described herein and its current implementation as part of LAPACK is different than the original LAPACK implementation described in Dongarra et al. [1989]. The original algorithm proposed in that paper cast even more computation in terms of matrix-vector product.

Every step (processing of a column block) of both algorithms consists of two stages: computation of multiple Householder transforms via unblocked algorithms (stage 1), and updating of the remaining data with blocked algorithms (stage 2). Stage 1 consists of the computation of several Householder reflectors, one for every column in the column block, and the updating of the rest of the column block, all of them via BLAS-2 operations. Stage 2 can be performed, all of it with BLAS-3 operations. Figure 4 shows the blocks that are touched (read/written) in these two stages for both algorithms. It can be easily checked that the new implementation uses more BLAS-3 computations than the LAPACK implementation. Moreover, the new implementation touches a

smaller part of the matrix in stage 1. As the right part of matrix A is read b times (once for every column being processed) in this stage, the amount of data brought to the processor by the new implementation from main memory is much smaller, thus reducing memory traffic.

4.4 Cost Analysis

The problem in all algorithms for reducing a matrix to upperHessenberg form is that some of the computation is in level 2 BLAS operations which attain only a fraction of the peak performance of current architectures. In particular, it is the matrix-vector product in (5) that contributes $O(n^3)$ flops aggregate over all iterations. It is the constant before n^3 that sets the algorithms in Sections 4.2 and 4.3 apart.

For the new algorithm proposed in Section 4.2, the matrix in (5) is roughly $(n-k)\times(n-k)$ during the iteration involving the kth column of A. The total number of flops in this operation, over all iterations, is approximately $2\sum_{k=0}^{n-k}(n-k)^2 \approx \frac{2}{3}n^3$. By contrast, in the LAPACK-like algorithm, the matrix in that computation spans all rows and is thus roughly $n\times(n-k)$. Combined over all iterations, the number of flops computed with matrix-vector products for the LAPACK-style algorithm is given by approximately $2\sum_{k=0}^{n-k} n(n-k) \approx n^3$. Recalling that the total cost of a reduction is about $\frac{10}{3}n^3$ flops, the new algorithm performs about 20% of its computation in level 2 BLAS and about 80% in level 3 BLAS. By contrast, the LAPACK-style algorithm spends about 30% in level 2 BLAS and about 70% in level 3 BLAS.

Even though most computation is in high-performing level 3 BLAS, the time spent in these matrix-vector products is often the dominant term since they are executed at a much lower rate. As a result, the reduction in the amount of computation performed in the matrix-vector products is significant as we will see in the performance reported in the next section.

5. EXPERIMENTS

We now demonstrate that, by shifting the computation from level 2 BLAS to level 3 BLAS operations, a noticeable performance improvement can be observed.

Five different implementations were tested. The first two correspond to routines DGEHD2 and DGEHRD from the LAPACK library and implement the unblocked algorithm and the LAPACK-style blocked algorithm, respectively. The other three (FLA_1, FLA_2, and FLA_3), implement the new blocked algorithms using the Formal Linear Algebra Methods Environment (FLAME) Application Programming Interface (API) for the C programming language. The FLAME APIs allow code to closely resemble the algorithms as they are illustrated in Figures 1–3. We refer the interested reader to other papers on FLAME [Bientinesi et al. 2005; Bientinesi et al. 2005]. FLA_1 is an implementation of the new algorithm using FLAME coding in both stage 1 (block computing) and stage 2 (block updating). FLA_2 employs index coding for first stage and FLAME coding for the second stage. FLA_3 is an implementation of the new algorithm using usual index coding in both stages.

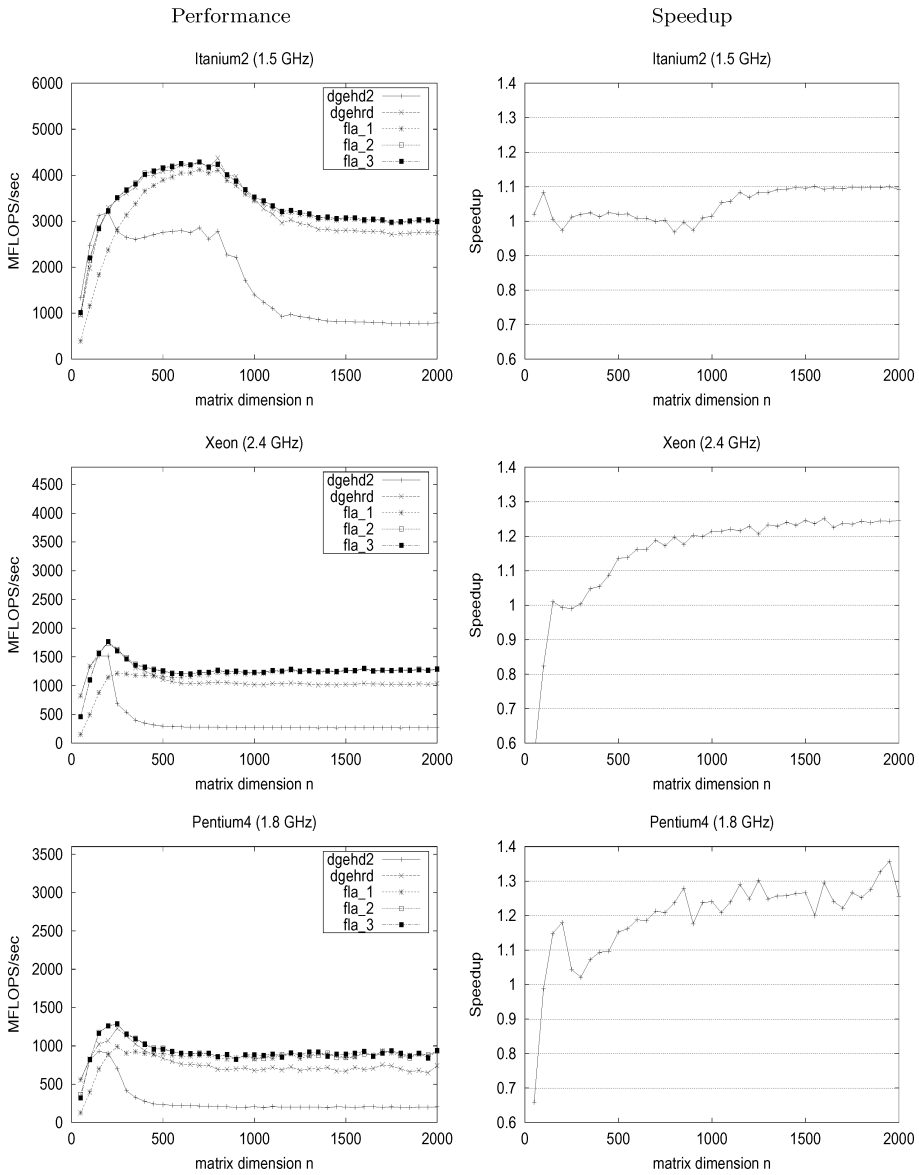


Fig. 5. Performance on various architectures as a function of matrix size, n, using a block size of $nb = 32$. When the matrix-vector product operation is relatively fast (on the Itanium 2), the overall performance is better and the speedup that is observed is less, since less time is spent in matrix-vector product. When the matrix-vector product is slow (on the Xeon and Pentium 4), the overall performance is worse, and the speedup gained by shifting computation from the matrix-vector product to the matrix-matrix product is more noticeable.

The implementations of the LAPACK-style and the new algorithm do not accumulate T. Rather, they accumulate its inverse, S. Moreover, in our implementation, this matrix S is combined with U so that the product US^T is accumulated in a matrix W: $I - UT^{-T}U^T$ becomes $I - WU^T$ and $VT^{-1}U^T$ becomes

VW^T. We note that, in the previous sections, we presented the algorithms using the UT transform since it is a more general way of stating the algorithm before these kinds of details are incorporated.

In Figure 5, performance is reported for the Xeon (2.4GHz), Pentium 4 (1.8GHz), and Itanium 2 (1.5GHz) processors. The Pentium 4 has two levels of cache, with a 512Kbyte L2 cache. The Xeon and Itanium 2 each have three levels of cache, with 1Mbyte and 6Mbyte L3 caches, respectively. On all platforms, BLAS libraries implemented by Kazushige Goto [2004] were used. Since, in this article, we are primarily concerned with demonstrating the benefits of the new algorithm rather than a complete study of the effect of blocking on different architectures, the block size was fixed at 32 for both blocked algorithms. From additional experiments, it was obvious that, for smaller problems, smaller blocksizes should be employed. On all three platforms, the new algorithm was noticeably faster than the one implemented in LAPACK. As can be expected, the difference was the least for the Itanium 2 processor which has a very fast and very large (6MBytes) L3 cache.

6. CONCLUSION

In this article, we have presented a new blocked algorithm for the reduction of a matrix to upperHessenberg form. While the new algorithm performs roughly the same number of computations as the algorithm that is currently included in LAPACK, it shifts more computation to high-performing matrix-matrix computations (level 3 BLAS). As a result, the overall performance of the computation is improved. The predicted improvement in performance was observed in practice on processors that are currently in common use.

We note that the new algorithm can be applied to a symmetric matrix, yielding a tridiagonal matrix. However, many operations can be saved by not updating the symmetric part stored in the upper triangular part of A. As a result, our observation cannot be used to speedup the LAPACK routine for reduction to tridiagonal form: the part of the matrix where the savings is attained is now not updated. The same observation applies to the LAPACK routine for reduction to bidiagonalization.

REFERENCES

ANDERSON, E., BAI, Z., DEMMEL, J., DONGARRA, J. E., DUCROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A. E., OSTROUCHOV, S., AND SORENSEN, D. 1999. *LAPACK Users' Guide 3rd Ed*. SIAM, Philadelphia, PA.

BARTELS, R. AND STEWART, G. 1972. Solution of the matrix equation $AX + XB = C$: Algorithm 432. *Commun. ACM 15*, 820–826.

BIENTINESI, P., GUNNELS, J. A., MYERS, M. E., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2005. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft. 31*, 1, 1–26.

BIENTINESI, P., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2005. Representing linear algebra algorithms in code: The FLAME APIs. *ACM Trans. Math. Soft. 31*, 1, 27–59.

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

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

DONGARRA, J. J., HAMMARLING, S. J., AND SORENSEN, D. C. 1989. Block reduction of matrices to condensed forms for eigenvalue computations. *J. Comput. Appl. Math. 27*.

GOLUB, G. H., NASH, S., AND VAN LOAN, C. F. 1979. A Hessenberg–Schur method for the problem $AX + XB = C$. *AC-24*, 909–913.

GOLUB, G. H. AND VAN LOAN, C. F. 1996. *Matrix Computations*, 3rd Ed. The Johns Hopkins University Press, Baltimore, MD.

GOTO, K. 2004. `http://www.cs.utexas.edu/users/kgoto`.

HOUSEHOLDER, A. S. 1958. Unitary triangularization of a nonsymmetric matrix. *J. ACM 5*, 339–42.

JOFFRAIN, T., LOW, T. M., QUINTANA-ORTÍ, E. S., VAN DE GEIJN, R., AND VAN ZEE, F. G. 2006. On accumulating householder transformations. *ACM Trans. Math. Soft. 32*, 2, 169–179.

MARTIN, R. S. AND WILKINSON, J. H. 1968. Similarity reduction of a general matrix to Hessenberg form. *Numer. Math. 12*, 349–68.

PUGLISI, C. 1992. Modification of the householder method based on the compact wy representation. *SIAM J. Sci. Stat. Comput. 13*, 723–726.

SCHREIBER, R. AND VAN LOAN, C. 1989. A storage-efficient WY representation for products of Householder transformations. *SIAM J. Sci. Stat. Comput. 10*, 1 (Jan.), 53–57.

SIMA, V. 1996. *Algorithms for Linear-Quadratic Optimization*. Pure and Applied Mathematics, vol. 200. Marcel Dekker, Inc., New York, NY.

WALKER, H. F. 1988. Implementation of the GMRES method using Householder transformations. *SIAM J. Sci. Stat. Comput. 9*, 1, 152–163.

WILKINSON, J. H. 1965. *The Algebraic Eigenvalue Problem*. Oxford University Press, Oxford, UK.

Received October 2004; revised July 2005; accepted August 2005

Anatomy of High-Performance Matrix Multiplication

KAZUSHIGE GOTO and ROBERT A. VAN DE GEIJN
The University of Texas at Austin

We present the basic principles that underlie the high-performance implementation of the matrix-matrix multiplication that is part of the widely used GotoBLAS library. Design decisions are justified by successively refining a model of architectures with multilevel memories. A simple but effective algorithm for executing this operation results. Implementations on a broad selection of architectures are shown to achieve near-peak performance.

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

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Linear algebra, matrix multiplication, basic linear algebra subprogrms

ACM Reference Format:

Goto, K. and van de Geijn, R. A. 2008. Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. 34, 3, Article 12 (May 2008), 25 pages. DOI = 10.1145/1356052.1356053 http://doi.acm.org/10.1145/1356052.1356053

1. INTRODUCTION

Implementing matrix multiplication so that near-optimal performance is attained requires a thorough understanding of how the operation must be layered at the macro level in combination with careful engineering of high-performance kernels at the micro level. This paper primarily addresses the macro issues, namely how to exploit a high-performance inner-kernel, more so than the the micro issues related to the design and engineering of that inner-kernel.

This research was sponsored in part by NSF grants ACI-0305163, CCF-0342369, and CCF-0540926, and by Lawrence Livermore National Laboratory project grant B546489.
We gratefully acknowledge equipment donations by Dell, Linux Networx, Virginia Tech, and Hewlett-Packard. Access to additional equipment was arranged by the Texas Advanced Computing Center and Lawrence Livermore National Laboratory.
Authors' addresses: K. Goto, Texas Advanced Computing Center, University of Texas at Austin, Austin, TX 78712; email: kgoto@tacc.utexas.edu; R. A. van de Geijn, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712; email: rvdg@cs.utexas.edu.

In Gunnels et al. [2001] a layered approach to the implementation of matrix multiplication was reported. The approach was shown to optimally amortize the required movement of data between two adjacent memory layers of an architecture with a complex multilevel memory. Like other work in the area [Agarwal et al. 1994; Whaley et al. 2001], Gunnels et al. [2001] casts computation in terms of an "inner-kernel" that computes $C := \tilde{A}B + C$ for some $m_c \times k_c$ matrix $\tilde{A}$ that is stored contiguously in some packed format and fits in cache memory. Unfortunately, the model for the memory hierarchy that was used is unrealistic in at least two ways:

—It assumes that this inner-kernel computes with a matrix $\tilde{A}$ that resides in the level-1 (L1) cache.

—It ignores issues related to the Translation Look-aside Buffer (TLB).

The current article expands upon a related technical report [Goto and van de Geijn 2002], which makes the observations that

—The ratio between the rate at which floating point operations (flops) can be performed by the floating point unit(s) and the rate at which floating point numbers can be streamed from the level-2 (L2) cache to registers is typically relatively small. This means that matrix $\tilde{A}$ can be streamed from the L2 cache.

—It is often the amount of data that can be addressed by the TLB that is the limiting factor for the size of $\tilde{A}$. (Similar TLB issues were discussed in Strazdins [1998].)

In addition, we now observe that

—There are in fact six inner-kernels that should be considered for building blocks for high-performance matrix multiplication. One of these is argued to be inherently superior over the others. (In Gunnel et al. [2001, 2005] three of these six kernels were identified.)

Careful consideration of all these observations underlie the implementation of the DGEMM Basic Linear Algebra Subprograms (BLAS) routine that is part of the widely used GotoBLAS library [Goto 2005].

In Figure 1 we preview the effectiveness of the techniques. In those graphs we report performance of our implementation as well as vendor implementations (Intel's MKL (8.1.1) and IBM's ESSL (4.2.0) libraries) and ATLAS [Whaley and Dongarra 1998] (3.7.11) on the Intel Pentium4 Prescott processor, the IBM Power 5 processor, and the Intel Itanium2 processor.[1] It should be noted that the vendor implementations have adopted techniques very similar to those described in this paper. It is important not to judge the performance of matrix-matrix multiplication in isolation. It is typically a building block for other operations like the level-3 BLAS (matrix-matrix operations) [Dongarra et al. 1990; Kågström et al. 1998] and LAPACK [Anderson et al. 1999]. How

[1]All libraries that were timed use assembly-coded inner-kernels (including ATLAS). Compiler options `-fomit-frame-pointer -O3 -funroll-all-loops` were used.

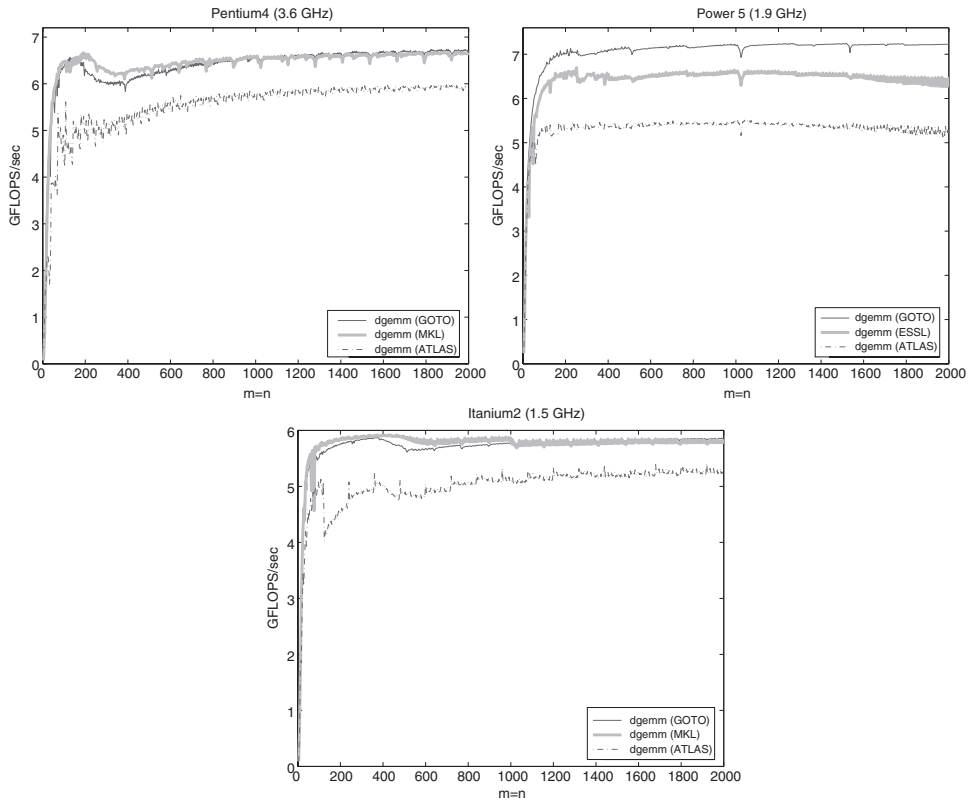


Fig. 1. Comparison of the matrix-matrix multiplication described in this paper with various other implementations. See Section 7 for details regarding the different architectures.

the techniques described in this article impact the implementation of level-3 BLAS is discussed in Goto and van de Geijn [2006].

This article attempts to describe the issues at a high level so as to make it accessible to a broad audience. Low level issues are introduced only as needed. In Section 2 we introduce notation that is used throughout the remainder of the article. In Section 3 a layered approach to implementing matrix multiplication is introduced. High-performance implementation of the inner-kernels is discussed in Section 4. Practical algorithms for the most commonly encountered cases of matrix multiplication are given in Section 5. In Section 6 we give further details that are used in practice to determine parameters that must be tuned in order to optimize performance. Performance results attained with highly tuned implementations on various architectures are given in Section 7. Concluding comments can be found in the final section.

2. NOTATION

The partitioning of matrices is fundamental to the description of matrix multiplication algorithms. Given an $m \times n$ matrix X, we will only consider

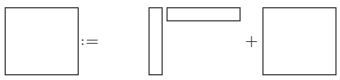
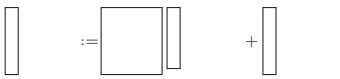
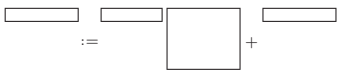
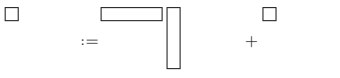
m	n	k	Illustration	Label
large	large	large		GEMM
large	large	small		GEPP
large	small	large		GEMP
small	large	large		GEPM
small	large	small		GEBP
large	small	small		GEPB
small	small	large		GEPDOT
small	small	small		GEBB

Fig. 2. Special shapes of GEMM $C := AB + C$. Here C, A, and B are $m \times n$, $m \times k$, and $k \times n$ matrices, respectively.

partitionings of X into blocks of columns and blocks of rows:

$$X = (X_0|X_1|\cdots|X_{N-1}) = \left(\begin{array}{c} \check{X}_0 \\ \hline \check{X}_1 \\ \hline \vdots \\ \hline \check{X}_{M-1} \end{array}\right),$$

where X_j has n_b columns and $\check{X}_i$ has m_b rows (except for X_{N-1} and $\check{X}_{M-1}$, which may have fewer columns and rows, respectively).

The implementations of matrix multiplication will be composed from multiplications with submatrices. We have given these computations special names, as tabulated in Figures 2 and 3. We note that these special shapes are very frequently encountered as part of algorithms for other linear algebra operations. For example, computation of various operations supported by LAPACK is cast mostly in terms of GEPP, GEMP, and GEPM. Even given a single dense linear algebra operation, multiple algorithms often exist where each of the algorithms casts the computation in terms of these different cases of GEMM multiplication [Bientinesi et al.].

Letter	Shape	Description
M	Matrix	Both dimensions are large or unknown.
P	Panel	One of the dimensions is small.
B	Block	Both dimensions are small.

Fig. 3. The labels in Figure 2 have the form GEXY where the letters chosen for X and Y indicate the shapes of matrices A and B, respectively, according to the above table. The exception to this convention is the GEPDOT operation, which is a generalization of the dot product.

3. A LAYERED APPROACH TO GEMM

In Figure 4 we show how GEMM can be decomposed systematically into the special cases that were tabulated in Figure 2. The general GEMM can be decomposed into multiple calls to GEPP, GEMP, or GEPM. These themselves can be decomposed into multiple calls to GEBP, GEPB, or GEPDOT kernels. The idea now is that if these three lowest level kernels attain high performance, then so will the other cases of GEMM. In Figure 5 we relate the path through Figure 4 that always take the top branch to a triple-nested loop. In that figure C, A, and B are partitioned into submatrices as

$$C = \begin{pmatrix} C_{11} & \cdots & C_{1N} \\ \vdots & & \vdots \\ C_{M1} & \cdots & C_{MN} \end{pmatrix}, \; A = \begin{pmatrix} A_{11} & \cdots & A_{1K} \\ \vdots & & \vdots \\ A_{M1} & \cdots & A_{MK} \end{pmatrix}, \text{ and } C = \begin{pmatrix} C_{11} & \cdots & C_{1N} \\ \vdots & & \vdots \\ C_{M1} & \cdots & C_{MN} \end{pmatrix},$$

where $C_{ij} \in \mathbb{R}^{m_c \times n_r}$, $A_{ip} \in \mathbb{R}^{m_c \times k_c}$, and $B_{pj} \in \mathbb{R}^{k_c \times n_r}$. The block sizes m_c, n_r, and k_c will be discussed in detail later in the paper.

A theory that supports an optimality claim regarding the general approach mentioned in this section can be found in Gunnels et al. [2001]. In particular, that paper supports the observation that computation should be cast in terms of the decision tree given in Figure 4 if data movement between memory layers is to be optimally amortized. However, the present article is self-contained, since, we show that the approach amortizes such overhead well and thus optimality is not crucial to our discussion.

4. HIGH-PERFORMANCE GEBP, GEPB, AND GEPDOT

We now discuss techniques for the high-performance implementation of GEBP, GEPB, and GEPDOT. We do so by first analyzing the cost of moving data between memory layers with an admittedly naive model of the memory hierarchy. In Section 4.2 we add more practical details to the model. This then sets the stage for algorithms for GEPP, GEMP, and GEPM in Section 5.

4.1 Basics

In Figure 6 (left) we depict a very simple model of a multilevel memory. One layer of cache memory is inserted between the Random-Access Memory (RAM) and the registers. The top-level issues related to the high-performance implementation of GEBP, GEPB, and GEPDOT can be described using this simplified architecture.

Let us first concentrate on GEBP with $A \in \mathbb{R}^{m_c \times k_c}$, $B \in \mathbb{R}^{k_c \times n}$, and $C \in \mathbb{R}^{m_c \times n}$. Partition

$$B = (B_0|B_1|\cdots|B_{N-1}) \quad \text{and} \quad C = (C_0|C_1|\cdots|C_{N-1})$$

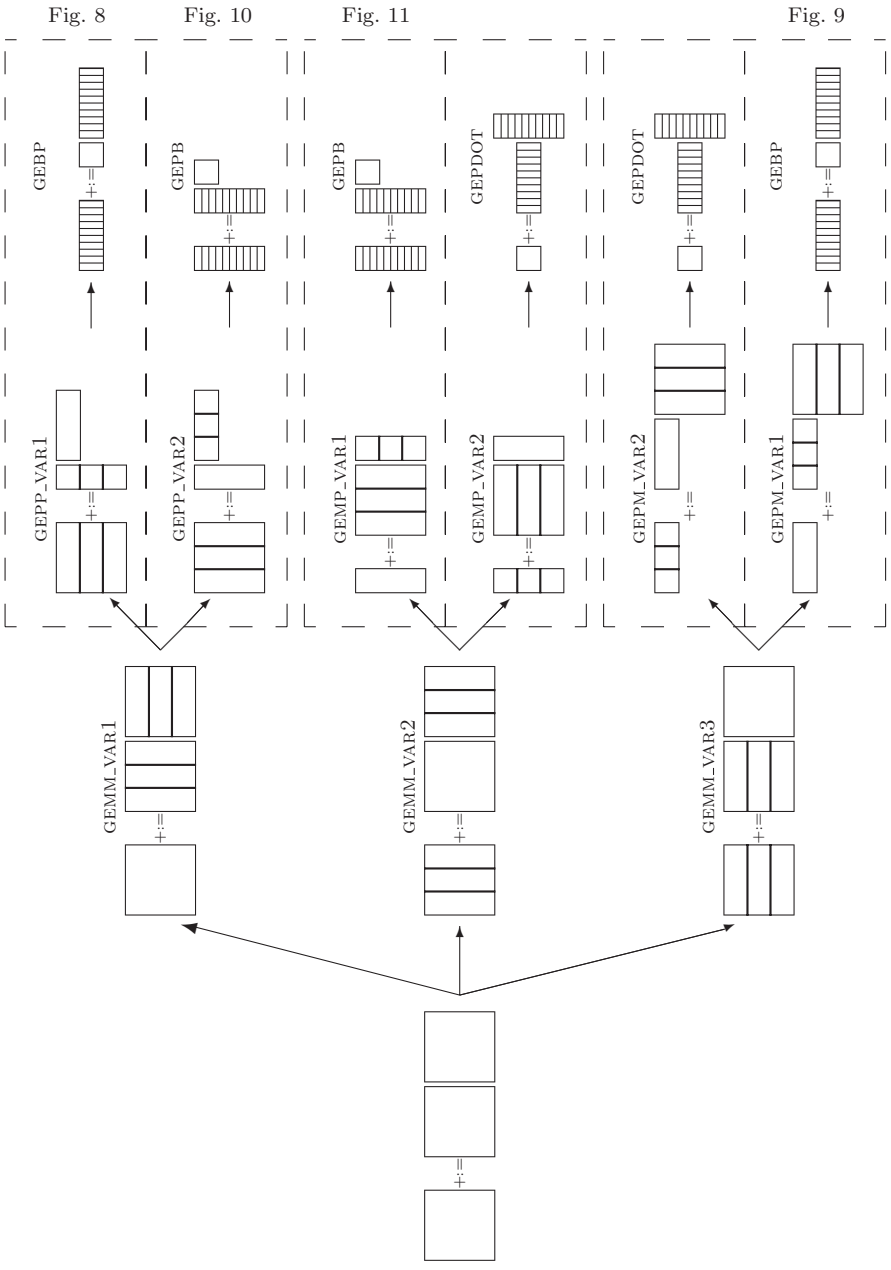


Fig. 4. Layered approach to implementing GEMM.

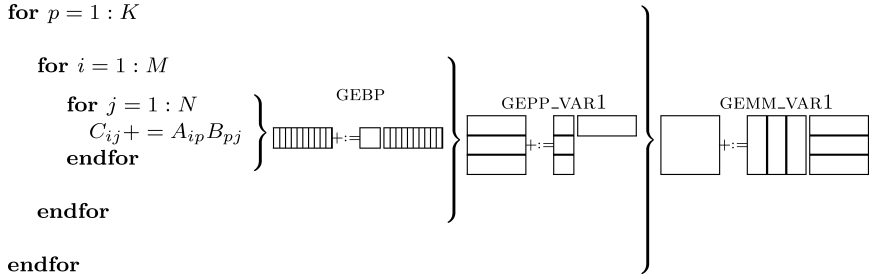


Fig. 5. The algorithm that corresponds to the path through Figure 4 that always takes the top branch expressed as a triple-nested loop.

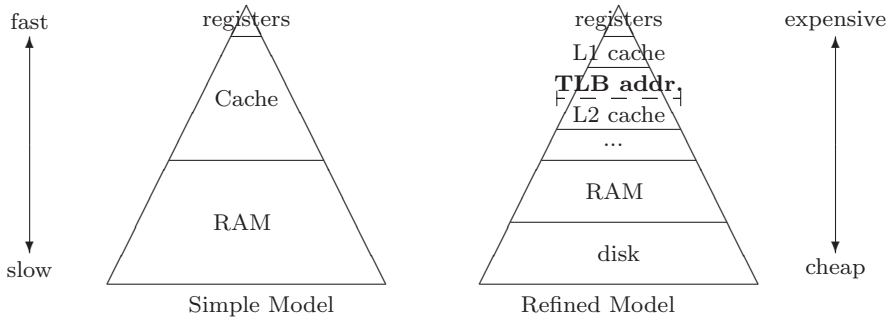


Fig. 6. The hierarchical memories viewed as a pyramid.

and assume that

Assumption (*a*). The dimensions m_c, k_c are small enough so that A and n_r columns from each of B and C (B_j and C_j, respectively) together fit in the cache.

Assumption (*b*). If A, C_j, and B_j are in the cache then $C_j := AB_j + C_j$ can be computed at the peak rate of the CPU.

Assumption (*c*). If A is in the cache it remains there until no longer needed.

Under these assumptions, the approach to GEBP in Figure 7 amortizes the cost of moving data between the main memory and the cache as follows. The total cost of updating C is $m_c k_c + (2m_c + k_c)n$ memops for $2m_c k_c n$ flops. Then the ratio between computation and data movement is

$$\frac{2m_c k_c n}{m_c k_c + (2m_c + k_c)n} \frac{\text{flops}}{\text{memops}} \approx \frac{2m_c k_c n}{(2m_c + k_c)n} \frac{\text{flops}}{\text{memops}} \quad \text{when } k_c \ll n. \tag{1}$$

Thus

$$\frac{2m_c k_c}{(2m_c + k_c)} \tag{2}$$

should be maximized under the constraint that $m_c k_c$ floating point numbers fill most of the cache, and the constraints in Assumptions (a)–(c). In practice there are other issues that influence the choice of k_c, as we will see in Section 6.3. However, the bottom line is that under the simplified assumptions A should

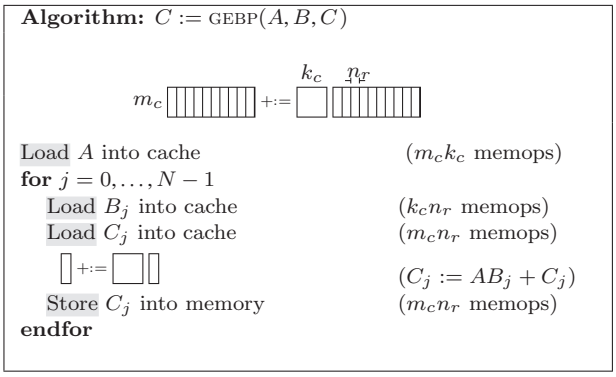


Fig. 7. Basic implementation of GEBP.

occupy as much of the cache as possible and should be roughly square,[2] while leaving room in the cache for at least B_j and C_j. If $m_c = k_c \approx n/100$ then even if memops are 10 times slower than flops, the memops add only about 10% overhead to the computation.

The related operations GEPB and GEPDOT can be analyzed similarly by keeping in mind the following pictures:

4.2 Refinements

In discussing practical considerations we will again focus on the high-performance implementation of GEBP. Throughout the remainder of the paper, we will assume that matrices are stored in column-major order.

4.2.1 *Choosing the Cache Layer.* A more accurate depiction of the memory hierarchy can be found in Figure 6(right). This picture recognizes that there are typically multiple levels of cache memory.

The first question is in which layer of cache the $m_c \times k_c$ matrix A should reside. Equation (2) tells us that (under Assumptions (a)–(c)) the larger $m_c \times n_c$, the better the cost of moving data between RAM and the cache is amortized over computation. This suggests loading matrix A in the cache layer that is farthest from the registers (can hold the most data) subject to the constraint that Assumptions (a)–(c) are (roughly) met.

The L1 cache inherently has the property that if it were used for storing A, B_j and C_j, then Assumptions (a)–(c) are met. However, the L1 cache tends to be very small. Can A be stored in the $L2$ cache instead, allowing m_c and k_c to be

[2]Note that optimizing the similar problem $m_c k_c/(2m_c + 2k_c)$ under the constraint that $m_c k_c \leq K$ is the problem of maximizing the area of a rectangle while minimizing the perimeter, the solution of which is $m_c = k_c$. We do not give an exact solution to the stated problem since there are practical issues that also influence m_c and k_c.

much larger? Let R_{comp} and R_{load} equal the rate at which the CPU can perform floating point operations and the rate at which floating point number can be streamed from the L2 cache to the registers, respectively. Assume A resides in the L2 cache and B_j and C_j reside in the L1 cache. Assume further that there is "sufficient bandwidth" between the L1 cache and the registers, so that loading elements of B_j and C_j into the registers is not an issue. Computing $AB_j + C_j$ requires $2m_c k_c n_r$ flops and $m_c k_c$ elements of A to be loaded from the L2 cache to registers. To overlap the loading of elements of A into the registers with computation $2n_r/R_{\text{comp}} \geq 1/R_{\text{load}}$ must hold, or

$$n_r \geq \frac{R_{\text{comp}}}{2R_{\text{load}}}. \tag{3}$$

4.2.2 *TLB Considerations.* A second architectural consideration relates to the page management system. For our discussion it suffices to consider that a typical modern architecture uses virtual memory so that the size of usable memory is not constrained by the size of the physical memory: Memory is partitioned into pages of some (often fixed) prescribed size. A table, referred to as the *page table*, maps virtual addresses to physical addresses and keeps track of whether a page is in memory or on disk. The problem is that this table itself is stored in memory, which adds additional memory access costs to perform virtual to physical translations. To overcome this, a smaller table, the Translation Look-aside Buffer (TLB), that stores information about the most recently used pages, is kept. Whenever a virtual address is found in the TLB, the translation is fast. Whenever it is not found (a TLB miss occurs), the page table is consulted and the resulting entry is moved from the page table to the TLB. In other words, the TLB is a cache for the page table. More recently, a level 2 TLB has been introduced into some architectures for reasons similar to those that motivated the introduction of an L2 cache.

The most significant difference between a cache miss and a TLB miss is that a cache miss does not necessarily stall the CPU. A small number of cache misses can be tolerated by using algorithmic prefetching techniques as long as the data can be read fast enough from the memory where it does exist and arrives at the CPU by the time it is needed for computation. A TLB miss, by contrast, causes the CPU to stall until the TLB has been updated with the new address. In other words, prefetching can mask a cache miss but not a TLB miss.

The existence of the TLB means that additional assumptions must be met:

Assumption (d). The dimensions m_c, k_c are small enough so that A, n_r columns from B (B_j) and n_r column from C (C_j) are simultaneously addressable by the TLB so that during the computation $C_j := AB_j + C_j$ no TLB misses occur.

Assumption (e). If A is addressed by the TLB, it remains so until no longer needed.

4.2.3 *Packing.* The fundamental problem now is that A is typically a submatrix of a larger matrix, and therefore is not contiguous in memory. This in turn means that addressing it requires many more than the minimal number

of TLB entries. The solution is to pack A in a contiguous work array, $\tilde{A}$. Parameters m_c and k_c are then chosen so that $\tilde{A}$, B_j, and C_j all fit in the L2 cache *and* are addressable by the TLB.

Case 1. *The TLB is the limiting factor*. Let us assume that there are T TLB entries available, and let $T_{\tilde{A}}$, T_{B_j}, and T_{C_j} equal the number of TLB entries devoted to $\tilde{A}$, B_j, and C_j, respectively. Then

$$T_{\tilde{A}} + 2(T_{B_j} + T_{C_j}) \leq T.$$

The reason for the factor two is that when the next blocks of columns B_{j+1} and C_{j+1} are first addressed, the TLB entries that address them should replace those that were used for B_j and C_j. However, upon completion of $C_j := \tilde{A}B_j + C_j$ some TLB entries related to $\tilde{A}$ will be the least recently used, and will likely be replaced by those that address B_{j+1} and C_{j+1}. The factor two allows entries related to B_j and C_j to coexist with those for $\tilde{A}$, B_{j+1} and C_{j+1} and by the time B_{j+2} and C_{j+2} are first addressed, it will be the entries related to B_j and C_j that will be least recently used and therefore replaced.

The packing of A into $\tilde{A}$, if done carefully, needs not to create a substantial overhead beyond what is already exposed from the loading of A into the L2 cache and TLB. The reason is as follows: The packing can be arranged so that upon completion $\tilde{A}$ resides in the L2 cache and is addressed by the TLB, ready for subsequent computation. The cost of accessing A to make this happen need not be substantially greater than the cost of moving A into the L2 cache, which is what would have been necessary even if A were not packed.

Operation GEBP is executed in the context of GEPP or GEPM. In the former case, the matrix B is reused for many separate GEBP calls. This means it is typically worthwhile to copy B into a contiguous work array, $\tilde{B}$, as well so that $T_{\tilde{B}_j}$ is reduced when $C := \tilde{A}\tilde{B} + C$ is computed.

Case 2. *The size of the L2 cache is the limiting factor*. A similar argument can be made for this case. Since the limiting factor is more typically the amount of memory that the TLB can address (e.g., the TLB on a current generation Pentium4 can address about 256Kbytes while the L2 cache can hold 2Mbytes), we do not elaborate on the details.

4.2.4 *Accessing Data Contiguously*. In order to move data most efficiently to the registers, it is important to organize the computation so that, as much as practical, data that is consecutive in memory is used in consecutive operations. One way to accomplish this is to not just pack A into work array $\tilde{A}$, but to arrange it carefully. We comment on this in Section 6.

From here on in this article, "Pack A into $\tilde{A}$" and "$C := \tilde{A}B + C$" will denote any packing that makes it possible to compute $C := AB + C$ while accessing the data consecutively, as much as needed. Similarly, "Pack B into $\tilde{B}$" will denote a copying of B into a contiguous format.

4.2.5 *Implementation of* GEPB *and* GEPDOT. Analyses of implementations of GEPB and GEPDOT can be similarly refined.

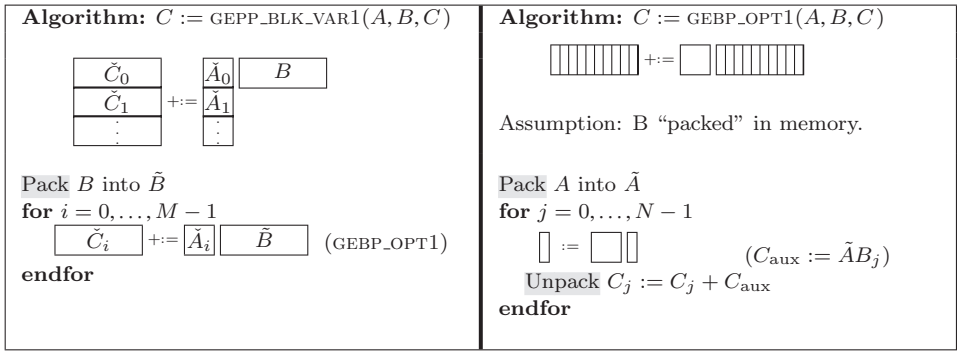


Fig. 8. Optimized implementation of GEPP (left) via calls to GEPB_OPT1 (right).

5. PRACTICAL ALGORITHMS

Having analyzed the approaches at a relatively coarse level of detail, we now discuss practical algorithms for all six options in Figure 4 while exposing additional architectural considerations.

5.1 Implementing GEPP with GEBP

The observations in Sections 4.2.1–4.2.4 are now summarized for the implementations of GEPP in terms of GEBP in Figure 8. The packing and computation are arranged to maximize the size of $\tilde{A}$: by packing B into $\tilde{B}$ in GEPP_VAR1, B_j typically requires only one TLB entry. A second TLB entry is needed to bring in B_{j+1}. The use of C_{aux} means that only one TLB entry is needed for that buffer, as well as up to n_r TLB entries for C_j (n_r if the leading dimension of C_j is large). Thus, $T_{\tilde{A}}$ is bounded by $T - (n_r + 3)$. The fact that C_j is not contiguous in memory is not much of a concern, since that data is not reused as part of the computation of the GEPP operation.

Once B and A have been copied into $\tilde{B}$ and $\tilde{A}$, respectively, the loop in GEBP_OPT1 can execute at almost the peak of the floating point unit.

—The packing of B is a memory-to-memory copy. Its cost is proportional to $k_c \times n$ and is amortized over $2m \times n \times k_c$ so that $O(m)$ computations will be performed for every copied item. This packing operation disrupts the previous contents of the TLB.

—The packing of A to $\tilde{A}$ rearranges this data from memory to a buffer that will likely remain in the L2 cache and leaves the TLB loaded with useful entries, if carefully orchestrated. Its cost is proportional to $m_c \times k_c$ and is amortized over $2m_c \times k_c \times n$ computation so that $O(n)$ computations will be performed for every copied item. In practice, this copy is typically less expensive.

This approach is appropriate for GEMM if m and n are both large, and k is not too small.

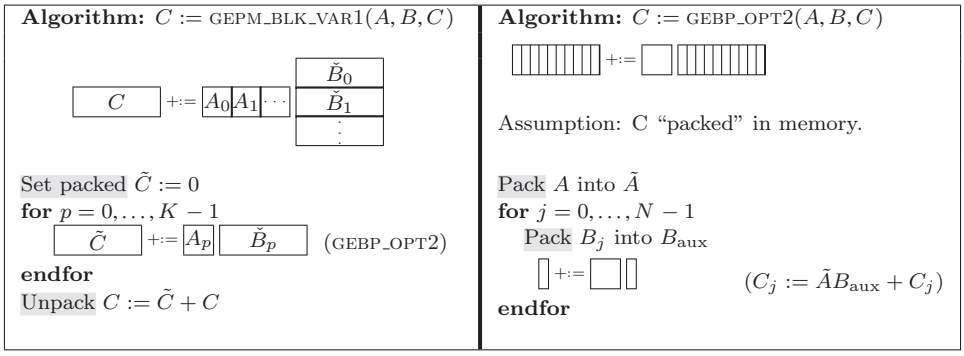


Fig. 9. Optimized implementation of GEPM (left) via calls to GEBP_OPT2 (right).

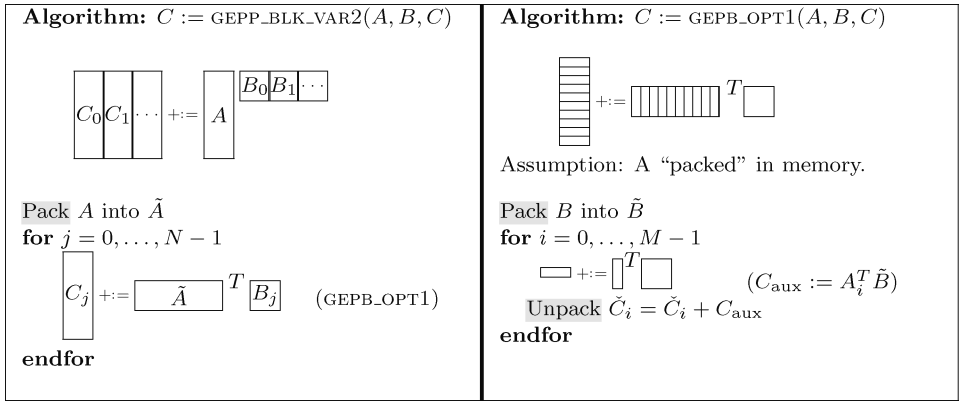


Fig. 10. Optimized implementation of GEMP (left) via calls to GEPB_OPT1 (right).

5.2 Implementing GEPM with GEBP

In Figure 9 a similar strategy is proposed for implementing GEPM in terms of GEBP. This time C is repeatedly updated so that it is worthwhile to accumulate $\tilde{C} = AB$ before adding the result to C. There is no reuse of $\check{B}_p$ and therefore it is not packed. Now at most n_r TLB entries are needed for B_j, and one each for B_{temp}, C_j and C_{j+1} so that again $T_{\tilde{A}}$ is bounded by $T - (n_r + 3)$.

5.3 Implementing GEPP with GEPB

Figure 10 shows how GEPP can be implemented in terms of GEPB. Now A is packed and transposed by GEPP to improve contiguous access to its elements. In GEPB B is packed and kept in the L2 cache, so that it is $T_{\tilde{B}}$ that we wish to maximize. While A_i, A_{i+1}, and C_{aux} each typically only require one TLB entry, $\check{C}_i$ requires n_c if the leading dimension of C is large. Thus, $T_{\tilde{B}}$ is bounded by $T - (n_c + 3)$.

5.4 Implementing GEMP with GEPB

In Figure 11 shows how GEMP can be implemented in terms of GEPB. This time a temporary $\tilde{C}$ is used to accumulate $\tilde{C} = (AB)^T$ and the L2 cache is mostly filled

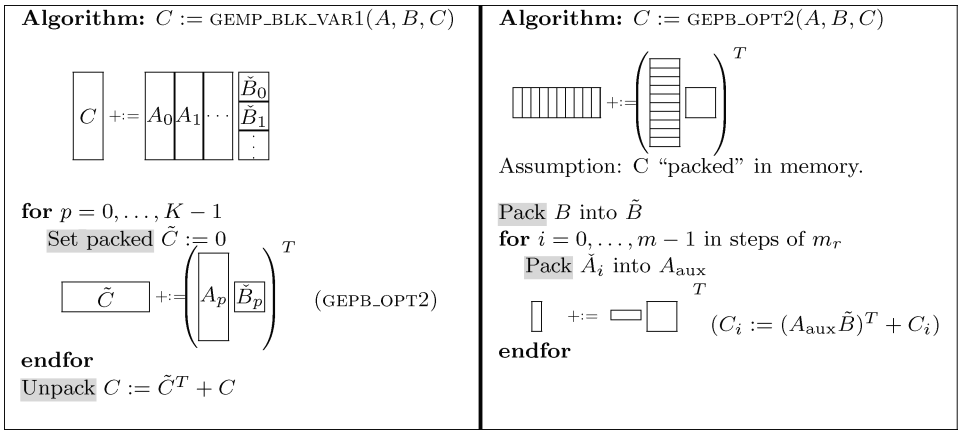


Fig. 11. Optimized implementation of GEPM (left) via calls to GEBP_OPT2 (right).

with a packed copy of $\tilde{B}$. Again, it is $T_{\tilde{B}}$ that we wish to maximize. While C_i, C_{i+1}, and A_{temp} each take up one TLB entry, $\check{A}_i$ requires up to m_c entries. Thus, $T_{\tilde{B}}$ is bounded by $T - (m_c + 3)$.

5.5 Implementing GEPM and GEMP with GEPDOT

Similarly, GEPM and GEMP can be implemented in terms of the GEPDOT operation. This places a block of C in the L2 cache and updates it by multiplying a few columns of A times a few rows of B. Since we will argue below that this approach is likely inferior, we do not give details here.

5.6 Discussion

Figure 4 suggests six different strategies for implementing a GEMM operation. Details for four of these options are given in Figures 8–11. We now argue that if matrices are stored in column-major order, then the approach in Figure 8, which corresponds to the path in Figure 4 that always takes the top branch in the decision tree and is also given in Figure 5, can in practice likely attain the best performance.

Our first concern is to attain the best possible bandwidth use from the L2 cache. Note that a GEPDOT-based implementation places a block of C in the L2 cache and reads *and writes* each of its elements as a few columns and rows of A and B are streamed from memory. This requires twice the bandwidth between the L2 cache and registers as do the GEBP and GEPBbased algorithms. Thus, we expect GEPDOT-based implementations to achieve worse performance than the other four options.

Comparing the pair of algorithms in Figures 8 and 9 the primary difference is that the former packs B and streams elements of C from and to main memory, while the latter streams B from main memory, computes C in a temporary buffer, and finally unpacks by adding this temporary result to C. In practice, the algorithm in Figure 8 can hide the cost of bringing elements of C from and to memory with computation while it exposes the packing of B as sheer

overhead. The algorithm in Figure 9 can hide the cost of bringing elements of B from memory, but exposes the cost of unpacking C as sheer overhead. The unpacking of C is a more complex operation and can therefore be expected to be more expensive than the packing of B, making the algorithm in Figure 8 preferable over the one in Figure 9. A similar argument can be used to rank the algorithm in Figure 10 over the one in Figure 11.

This leaves us with having to choose between the algorithms in Figures 8 and 10, which on the surface appear to be symmetric in the sense that the roles of A and B are reversed. Note that the algorithms access C a few columns and rows at a time, respectively. If the matrices are stored in column-major order, then it is preferable to access a block of those matrices by columns. Thus the algorithm in Figure 8 can be expected to be superior to all the other presented options.

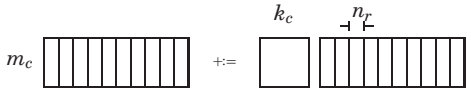
Due to the level of effort that is required to implement kernels like GEBP, GEPB, and GEPDOT, we focus on the algorithm in Figure 8 throughout the remainder of this article.

We stress that the conclusions in this subsection are continguent on the observation that on essentially all current processors there is an advantage to blocking for the L2 cache. It is entirely possible that the insights will change if, for example, blocking for the L1 cache is preferred.

6. MORE DETAILS YET

We now give some final insights into how registers are used by kernels like GEBP_OPT1, after which we comment on how parameters are chosen in practice.

Since it has been argued that the algorithm in Figure 8 will likely attain the best performance, we focus on that algorithm:



6.1 Register Blocking

Consider $C_{\text{aux}} := \tilde{A}B_j$ in Figure 8, where $\tilde{A}$ and B_j are in the L2 and L1 caches, respectively. This operation is implemented by computing $m_r \times n_r$ submatrices of C_{aux} in the registers.

Notice that this means that during the computation of C_j it is not necessary that elements of that submatrix remain in the L1 or even the L2 cache: $2m_r n_r k_c$ flops are performed for the $m_r n_r$ memops that are needed to store the results from the registers to whatever memory layer. We will see that k_c is chosen to be relatively large.

This figure allows us to discuss the packing of A into $\tilde{A}$ in more detail. In our implementation, $\tilde{A}$ is stored so that each $m_r \times k_c$ submatrix is stored contiguously in memory. Each such submatrix is itself stored in column-major order. This allows $C_{\text{aux}} := \tilde{A}B_j$ to be computed while accessing the elements of $\tilde{A}$ by

striding strictly contiguously through memory. Implementations by others will often store $\tilde{A}$ as the transpose of A, which requires a slightly more complex pattern when accessing $\tilde{A}$.

6.2 Choosing $m_r \times n_r$

The following considerations affect the choice of $m_r \times n_r$:

—Typically half the available registers are used for the storing $m_r \times n_r$ submatrix of C. This leaves the remaining registers for prefetching elements of $\tilde{A}$ and $\tilde{B}$.

—It can be shown that amortizing the cost of loading the registers is optimal when $m_r \approx n_r$.

—As mentioned in Section 4.2.1, the fetching of an element of $\tilde{A}$ from the L2 cache into registers must take no longer than the computation with a previous element so that ideally $n_r \geq R_{\text{comp}}/(2R_{\text{load}})$ must hold. R_{comp} and R_{load} can be found under "flops/cycle" and "Sustained Bandwidth", respectively, in Figure 12.

A shortage of registers will limit the performance that can be attained by GEBP_OPT1, since it will impair the ability to hide constraints related to the bandwidth to the L2 cache.

6.3 Choosing k_c

To amortize the cost of updating $m_r \times n_r$ elements of C_j the parameter k_c should be picked to be as large as possible.

The choice of k_c is limited by the following considerations:

—Elements from B_j are reused many times, and therefore must remain in the L1 cache. In addition, the set associativity and cache replacement policy further limit how much of the L1 cache can be occupied by B_j. In practice, $k_c n_r$ floating point numbers should occupy less than half of the L1 cache so that elements of $\tilde{A}$ and C_{aux} do not evict elements of B_j.

—The footprint of $\tilde{A}$ ($m_c \times k_c$ floating point numbers) should occupy a considerable fraction of the L2 cache.

In our experience the optimal choice is such that k_c double precision floating point numbers occupy half of a page. This choice typically satisfies the other constraints as well as other architectural constraints that go beyond the scope of this article.

6.4 Choosing m_c

It was already argued that $m_c \times k_c$ matrix $\tilde{A}$ should fill a considerable part of the smaller of (1) the memory addressable by the TLB and (2) the L2 cache. In fact, this is further constrained by the set-associativity and replacement policy of the L2 cache. In practice, m_c is typically chosen so that $\tilde{A}$ only occupies about half of the smaller of (1) and (2).

7. EXPERIMENTS

In this section we report performance attained by implementations of the DGEMM BLAS routine using the techniques described in the earlier sections. It is *not* the purpose of this section to show that our implementations attain better performance than those provided by vendors and other projects. (We do note, however, that they are highly competitive.) Rather, we attempt to demonstrate that the theoretical insights translate to practical implementations.

7.1 Algorithm Chosen

Implementing all algorithms discussed in Section 5 on a cross-section of architectures would be a formidable task. Since it was argued that the approach in Figure 8 is likely to yield the best overall performance, it is that variant which was implemented. The GEBP_opt1 algorithm was carefully assembly-coded for each of the architectures that were considered. The routines for packing A and B into $\tilde{A}$ and $\tilde{B}$, respectively, were coded in C, since compilers appear to be capable of optimizing these operations.

7.2 Parameters

In Figure 12 we report the physical and algorithmic parameters for a cross-section of architectures. Not all the parameters are taken into account in the analysis in this paper. These are given for completeness.

The following parameters require extra comments.

Duplicate. This parameter indicates whether elements of matrix B are duplicated as part of the packing of B. This is necessary in order to take advantage of SSE2 instructions on the Pentium4 (Northwood) and Opteron processors. Although the Core 2 Woodcrest has an SSE3 instruction set, instructions for duplication are issued by the multiply unit and the same techniques as for the Northwood architecture must be employed.

Sustained Bandwidth. This is the *observed* sustained bandwidth, in doubles/cycle, from the indicated memory layer to the registers.

Covered Area. This is the size of the memory that can be addressed by the TLB. Some architectures have a (much slower) level 2 TLB that serves the same function relative to an L1 TLB as does an L2 cache relative to an L1 cache. Whether to limit the size of $\tilde{A}$ by the number of entries in L1 TLB or L2 TLB depends on the cost of packing into $\tilde{A}$ and $\tilde{B}$.

$\tilde{A}$ (*Kbytes*). This indicates how much memory is set aside for matrix $\tilde{A}$.

7.3 Focus on the Intel Pentium 4 Prescott Processor (3.6 GHz, 64bit)

We discuss the implementation for the Intel Pentium 4 Prescott processor in greater detail. In Section 7.4 we more briefly discuss implementations on a few other architectures.

Equation (3) indicates that in order to hide the prefetching of elements of $\tilde{A}$ with computation parameter n_r must be chosen so that $n_r \geq R_{\text{comp}}/(2R_{\text{load}})$. Thus, for this architecture, $n_r \geq 2/(2 \times 1.03) \approx 0.97$. Also, EM64T architectures,

Architecture			L1 cache					L2 cache					L3 cache				TLB			Block sizes	
Sub Architecture	Core	# of registers	flops/cycle	Duplicate	Size (Kbytes)	Line Size	Associativity	Sustained Bandwidth	Size (Kbytes)	Line Size	Associativity	Sustained Bandwidth	Size (Kbytes)	Line Size	Associativity	Sustained Bandwidth	Page Size (Kbytes)	L1 TLB	L2 TLB	Covered Area (Kbytes)	$\hat{A}$ (Kbytes)	$m_c \times k_c$	$m_r \times n_r$
x86																							
Pentium3	Katmai	8	1	N	16	32	4	0.95	512	32	4	0.40					4	64	-	256		64×256	2×2
	Coppermine	8	1	N	16	32	4	0.95	256	32	4	0.53					4	64	-	256		64×256	2×2
Pentium4	Northwood	8[1]	2	Y	8	64	4	1.88	512	64	8	1.06					4	64	-	256	224	224×128	4×2
	Prescott	8[1]	2	N	16	64	4	1.92	2K	64	8	1.03					4	64	-	256	768	768×128	2×4
Opteron		8[1]	2	Y	64	64	2	2.00	1K	64	16	0.71					4	32	512	$2K^2$	768	384×256	2×4
x86_64 (EM64T)																							
Pentium4	Prescott	16[1]	2	N	16	64	4	1.92	2K	64	8	1.03					4	64	-	256	$\approx$ 1K	696×192	4×4
Core 2	Woodcrest	16[1]	4	Y	32	64	8	2.00	4K	64	8	1.00					4	256	-	1K	1K	512×256	4×4
Opteron		16[1]	2	Y	64	64	2	2.00	1K	64	16	0.71					4	32	512	$2K^2$	608	384×256	4×4
IA64																							
Itanium2		128	4	N	16	64	4	-	256	128	8	4.00	6K	128	24	2.00	16	-	128	$2K^2$	1K	$128 \times 1K$	8×8
POWER																							
POWER3		32	4	N	64	128	4	2.00	1K	128	1	0.75					4	256	-	1K	512	256×256	4×4
POWER4		32	4	N	32	128	2	1.95	1.4K	128	4	0.93	128K	512	8	-	4	128^4	$1K^4$	$4K^2$	288	144×256	4×4
PPC970		32	4	N	32	128	2	2.00	512	128	8	0.92					4	128^4	$1K^4$	$4K^2$	320	160×256	4×4
POWER5		32	4	N	32	128	2	2.00	1.92K	128	10	0.93		128			4	128^4	$1K^4$	$4K^2$	512	256×256	4×4
PPC440 FP2		32[1]	4	N	32	32	64	2.00	2	128	Full	0.75	4K	128	8	0.75	-[3]	-[3]	-[3]	-[3]	3K	$128 \times 3K$	8×4
Alpha																							
EV4		32	1	N	16	32	1	0.79	2K	32	1	0.15					8	32	-	256	14	32×56	4×4
EV5		32	2	N	8	32	1	1.58	96	64	3	1.58	2K	64	1	0.22	8	64	-	512	63	56×144	4×4
EV6		32	2	N	64	64	2	1.87	4K	64	1	0.62					8	128	-	1K	512	128×512	4×4
SPARC																							
IV		32	4	N	64	32	4	0.99	8K	128	2	0.12	8K	128	2	0.12	8	16	512	$4K^2$	2K	512 $\times$ 512	4 $\times$ 4

[1] Registers hold 2 floating point numbers each. [2] indicates that the Covered Area is determined from the L2 TLB. [3] IBM has not not disclosed this information. [4] On these machines there is a D-ERAT (Data cache Effective Real to Address Translation [table]) that takes the place of the L1 TLB and a TLB that acts like an L2 TLB.

Fig. 12. Parameters for a sampling of current architectures.

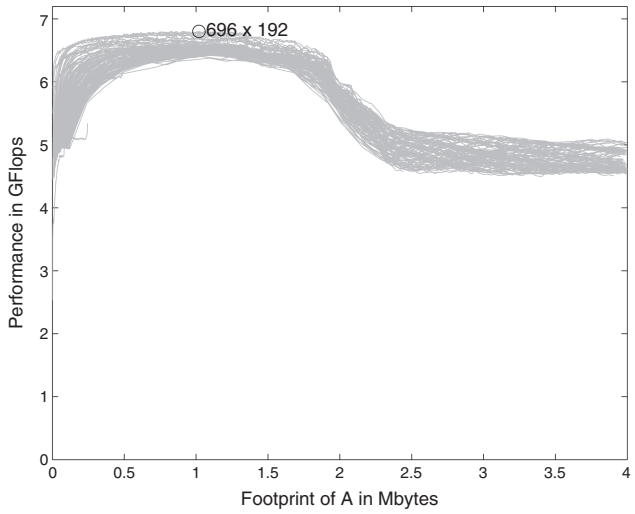


Fig. 13. Performance on the Pentium4 Prescott (3.6 GHz) of DGEMM for different choices of m_c and k_c, where m_c and k_c are varied from 8 to 2000 in steps of 8. This architecture has a 2Mbyte L2 cache, which explains the performance degradation when the footprint of $\breve{A}$ is about that size. The best performance is attained for $m_c \times k_c = 696 \times 196$, when the footprint is around 1 Mbyte.

of which this Pentium4 is a member, have 16 registers that can store two double precision floating point numbers each. Eight of these registers are used for storing entries of C: $m_r \times n_r = 4 \times 4$.

The choice of parameter k_c is complicated by the fact that updating the indexing in the loop that computes inner products of columns of $\tilde{A}$ and $\tilde{B}$ is best avoided on this architecture. As a result, that loop is completely unrolled, which means that storing the resulting code in the instruction cache becomes an issue, limiting k_c to 192. This is slightly smaller than the $k_c = 256$ that results from the limitation discussed in Section 6.3.

In Figure 13 we show the performance of DGEMM for different choices of m_c and k_c.[3] This architecture is the one exception to the rule that $\tilde{A}$ should be addressable by the TLB, since a TLB miss is less costly than on other architectures. When $\tilde{A}$ is chosen to fill half of the L2 cache the performance is slightly better than when it is chosen to fill half of the memory addressable by the TLB.

The Northwood version of the Pentium4 relies on SSE2 instructions to compute two flops per cycle. This instruction requires entries in B to be duplicated, a data movement that is incorporated into the packing into buffer $\tilde{B}$. The SSE3 instruction supported by the Prescott subarchitecture does not require this duplication when copying to $\tilde{B}$.

In Figure 14 we show the performance attained by the approach on this Pentium 4 architecture. In this figure the top graph shows the case where all matrices are square while the bottom graph reports the case where $m =$

[3]Ordinarily examining the performance for all possible combinations of m_c and k_c is not part of our optimization process. Rather, the issues discussed in this paper are used to identify a few possible combinations of parameters, and only combinations of m_c and k_c near these candidate parameters are tried. The graph is included to illustrate the effect of picking different parameters.

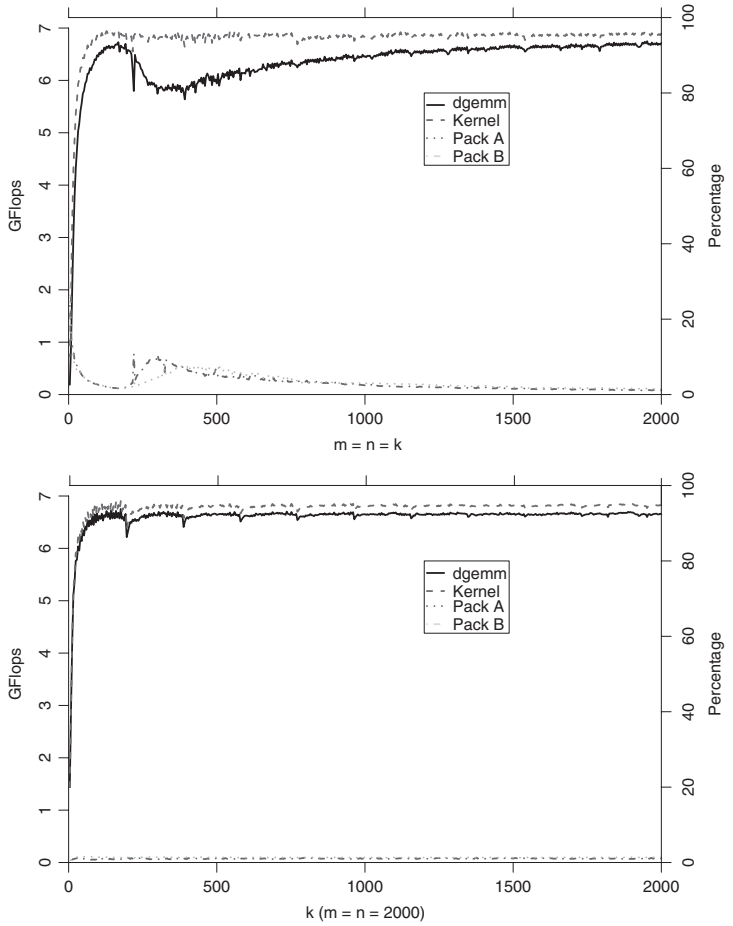


Fig. 14. Pentium4 Prescott (3.6 GHz).

$n = 2000$ and k is varied. We note that GEPP with k relatively small is perhaps the most commonly encountered special case of GEMM.

—The top curve, labeled Kernel, corresponds to the performance of the kernel routine (GEBP_opt1).

—The next lower curve, labeled dgemm, corresponds to the performance of the DGEMM routine implemented as a sequence of GEPP operations. The GEPP operation was implemented via the algorithms in Figure 8.

—The bottom two curves correspond the percent of time incurred by routines that pack A and B into $\tilde{A}$ and $\tilde{B}$, respectively. (For these curves only the labeling along the right axis is relevant.)

The overhead caused by the packing operations accounts almost exactly for the degradation in performance from the kernel curve to the DGEMM curve.

The graphs in Figure 15 investigate the performance of the implementation when m and n are varied. In the top graph m is varied while $n = k = 2000$.

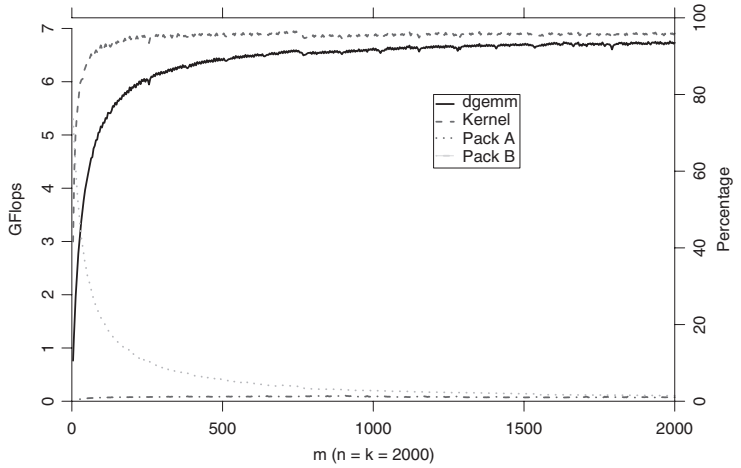


Fig. 15. Pentium4 Prescott(3.6 GHz).

When m is small, as it would be for a GEPM operation, the packing of B into $\tilde{B}$ is not amortized over sufficient computation, yielding relatively poor performance. One solution would be to skip the packing of B. Another would be to implement the algorithm in Figure 9. Similarly, in the bottom graph n is varied while $m = k = 2000$. When n is small, as it would be for a GEMP operation, the packing of A into $\tilde{A}$ is not amortized over sufficient computation, again yielding relatively poor performance. Again, one could contemplate skipping the packing of A (which would require the GEBP operation to be cast in terms of AXPY operations instead of inner-products). An alternative would be to implement the algorithm in Figure 11.

7.4 Other Architectures

For the remaining architectures we discuss briefly how parameters are selected and show performance graphs, in Figs. 16–20, that correspond to those for the Pentium 4 in Figure 14.

AMD Opteron processor (2.2 GHz, 64bit). For the Opteron architecture $n_r \geq R_{\text{comp}}/(2R_{\text{load}}) = 2/(2 \times 0.71) \approx 1.4$. The observed optimal choice for storing entries of C in registers is $m_r \times n_r = 4 \times 4$.

Unrolling of the inner loop that computes the inner-product of columns of $\tilde{A}$ and $\tilde{B}$ is not necessary like it was for the Pentium4, nor is the size of the L1 cache an issue. Thus, k_c is taken so that a column of $\tilde{B}$ fills half a page: $k_c = 256$. By taking $m_c \times k_c = 384 \times 256$ matrix $\tilde{A}$ fills roughly one third of the space addressable by the TLB.

The latest Opteron architectures support SSE3 instructions, we have noticed that duplicating elements of $\tilde{B}$ is still beneficial. This increases the cost of packing into $\tilde{B}$, decreasing performance by about 3%.

Performance for this architecture is reported in Figure 16.

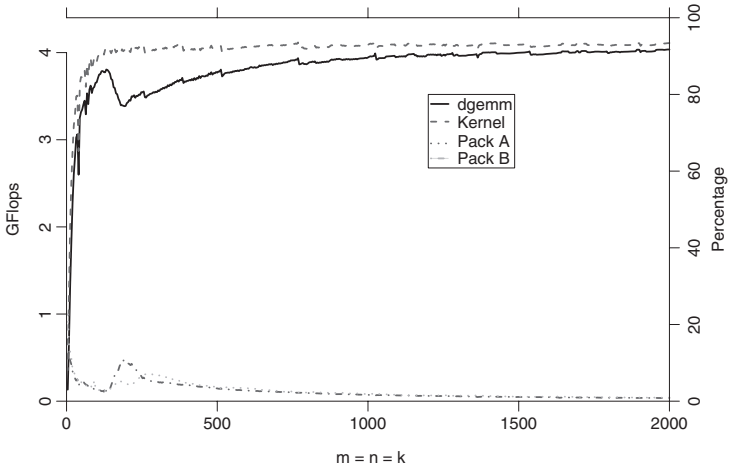


Fig. 16. Opteron (2.2 GHz).

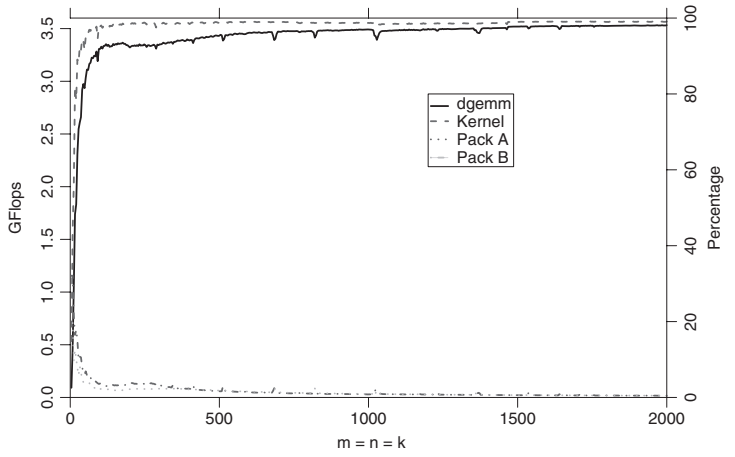


Fig. 17. Itanium2 (1.5 GHz).

Intel Itanium2 processor (1.5 GHz). The L1 data cache and L1 TLB are inherently ignored by this architecture for floating point numbers. As a result, the Itanium2's L2 and L3 caches perform the role of the L1 and L2 caches of other architectures and only the L2 TLB is relevant. Thus $n_r \geq 4/(2\times 2.0) = 1.0$. Since there are ample registers available, $m_r \times n_r = 8 \times 8$. While the optimal $k_c = 1$K (1K doubles fill half of a page), in practice performance is almost as good when $k_c = 128$.

This architecture has many features that makes optimization easy: A very large number of registers, very good bandwidth between the caches and the registers, and an absence of out-of-order execution.

Performance for this architecture is reported in Figure 17.

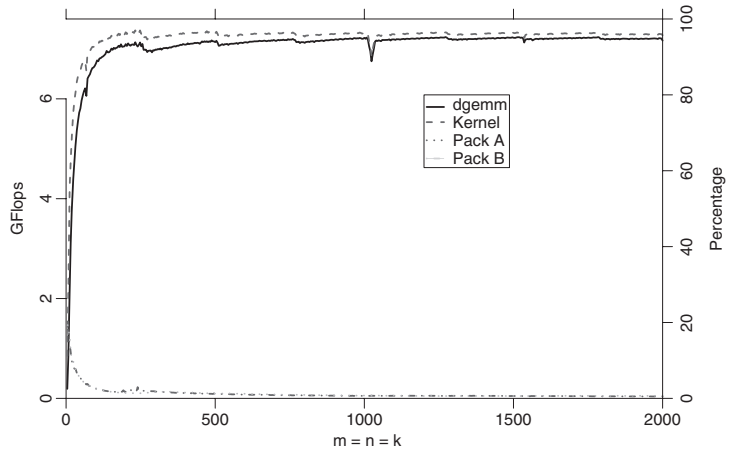


Fig. 18. POWER5 (1.9 GHz).

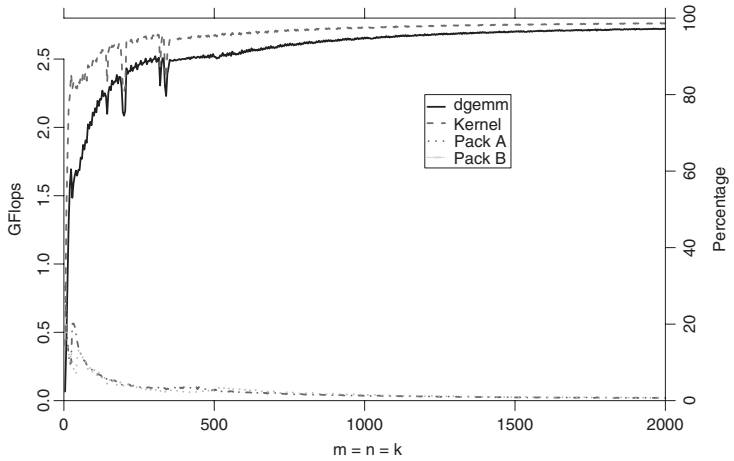


Fig. 19. PPC440 FP2 (700 MHz).

IBM POWER5 processor (1.9 GHz). For this architecture, $n_r \geq 4/(2\times 0.93) \approx 2.15$ and $m_r \times n_r = 4\times 4$. This architectures has a D-ERAT (Data cache Effective Real to Address Translation [table]) that acts like an L1 TLB and a TLB that acts like an L2 TLB. Parameter $k_c = 256$ fills half of a page with a column of $\tilde{B}$. By choosing $m_c \times k_c = 256 \times 256$ matrix $\tilde{A}$ fills about a quarter of the memory addressable by the TLB. This is a compromise: The TLB is relatively slow. By keeping the footprint of $\tilde{A}$ at the size that is addressable by the D-ERAT, better performance has been observed.

Performance for this architecture is reported in Figure 18.

PowerPC440 FP2 processor (700 MHz). For this architecture, $n_r \geq 4/(2 \times 0.75) \approx 2.7$ and $m_r \times n_r = 8 \times 4$. An added complication for this architecture is that the combined bandwidth required for moving elements of $\tilde{B}$ and $\tilde{A}$ from the L1 and L2 caches to the registers saturates the total bandwidth. This means

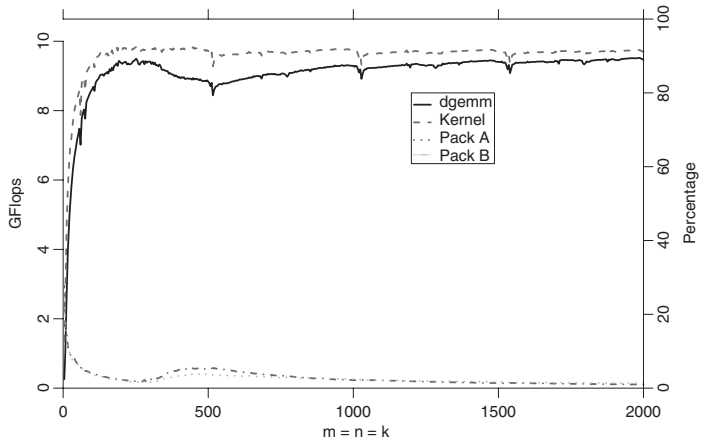


Fig. 20. Core 2 Woodcrest (2.66 GHz).

that the loading of elements of C into the registers cannot be overlapped with computation, which in turn means that k_c should be taken to be very large in order to amortize this exposed cost over as much computation as possible. The choice $m_c \times n_c = 128 \times 3K$ fills 3/4 of the L2 cache.

Optimizing for this architecture is made difficult by lack of bandwidth to the caches, an L1 cache that is FIFO (First-In-First-Out) and out-of-order execution of instructions. The addressable space of the TLB is large due to the large page size.

It should be noted that techniques similar to those discussed in this paper were used by IBM to implement their matrix multiply [Bachega et al. 2004] for this architecture.

Performance for this architecture is reported in Figure 19.

Core 2 Woodcrest (2.66 GHz) processsor. At the time of the final revision of this paper, the Core 2 Woodcrest was recently released and thus performance numbers for this architecture are particularly interesting.

For this architecture, $n_r \geq 4/(2 \times 1.0) = 2$ and $m_r \times n_r = 4 \times 4$. As for the Prescott architecture, sixteen registers that can hold two double precision numbers each are available, half of which are used to store the $m_r \times n_r$ entries of C. The footprint of matrix $\tilde{A}$ equals that of the memory covered by the TLB.

Performance for this architecture is reported in Figure 20.

8. CONCLUSION

We have given a systematic analysis of the high-level issues that affect the design of high-performance matrix multiplication. The insights were incorporated in an implementation that attains extremely high performance on a variety of architectures.

Almost all routines that are currently part of LAPACK [Anderson et al. 1999] perform the bulk of computation in GEPP, GEMP, or GEPM operations. Similarly, the important Basic Linear Algebra Subprograms (BLAS) kernels can be cast

in terms of these three special cases of GEMM [Kågström et al. 1998]. Our recent research related to the FLAME project shows how for almost all of these routines there are algorithmic variants that cast the bulk of computation in terms of GEPP [Gunnels et al. 2001; Bientinesi et al. 2005a; Bientinesi et al. 2005b; Low et al. 2005; Quintana et al. 2001]. These alternative algorithmic variants will then attain very good performance when interfaced with matrix multiplication routines that are implemented based on the insights in this paper.

One operation that cannot be recast in terms of mostly GEPP is the QR factorization. For this factorization, about half the computation can be cast in terms of GEPP while the other half inherently requires either the GEMP or the GEPM operation. Moreover, the panel must inherently be narrow since the wider the panel, the more extra computation must be performed. This suggests that further research into the high-performance implementation of these special cases of GEMM is warranted.

The source code for the discussed implementations is available from `http://www.tacc.utexas.edu/resources/software`.

ACKNOWLEDGMENTS

We would like to thank Victor Eijkhout, John Gunnels, Gregorio Quintana, and Field Van Zee for comments on drafts of this paper, as well as the anonymous referees.

REFERENCES

AGARWAL, R., GUSTAVSON, F., AND ZUBAIR, M. 1994. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM J. Resear. Devel. 38,* 5 (Sept.).

ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S., DEMMEL, J., DONGARRA, J., CROZ, J. D., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORENSEN, D. 1999. *LAPACK Users' Guide*, 3rd Ed. SIAM Press.

BACHEGA, L., CHATTERJEE, S., DOCKSER, K. A., GUNNELS, J. A., GUPTA, M., GUSTAVSON, F. G., LAPKOWSKI, C. A., LIU, G. K., MENDELL, M. P., WAIT, C. D., AND WARD, T. J. C. 2004. A high-performance simd floating point unit for bluegene/l: Architecture, compilation, and algorithm design. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE Computer Society, 85–96.

BIENTINESI, P., GUNNELS, J. A., MYERS, M. E., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2005a. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Softw. 31,* 1, 1–26.

BIENTINESI, P., GUNTER, B., AND VAN DE GEIJN, R. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw.*. To appear.

BIENTINESI, P., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2005b. Representing linear algebra algorithms in code: The FLAME APIs. *ACM Trans. Math. Softw. 31,* 1, 27–59.

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

GOTO, K. 2005. `www.tacc.utexas.edu/resources/software/`.

GOTO, K. AND VAN DE GEIJN, R. 2006. High-performance implementation of the level-3 BLAS. FLAME Working Note #20, Tech. rep. TR-2006-23, Department of Computer Sciences, The University of Texas at Austin.

GOTO, K. AND VAN DE GEIJN, R. A. 2002. On reducing TLB misses in matrix multiplication. Tech. rep. CS-TR-02-55, Department of Computer Sciences, The University of Texas at Austin.

GUNNELS, J., GUSTAVSON, F., HENRY, G., AND VAN DE GEIJN, R. A. 2005. A family of high-performance matrix multiplication algrorithms. Lecture Notes in Computer Science, vol. 3732. Springer-Verlag, 256–265.

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

GUNNELS, J. A., HENRY, G. M., AND VAN DE GEIJN, R. A. 2001. A family of high-performance matrix multiplication algorithms. In *Proceedings of the International Conference on Computational Science (ICCS'01), Part I*, V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. K. Tan, Eds. Lecture Notes in Computer Science, vol. 2073. Springer-Verlag, 51–60.

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

LOW, T. M., VAN DE GEIJN, R., AND VAN ZEE, F. 2005. Extracting SMP parallelism for dense linear algebra algorithms from high-level specifications. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*.

QUINTANA, E. S., QUINTANA, G., SUN, X., AND VAN DE GEIJN, R. 2001. A note on parallel matrix inversion. *SIAM J. Sci. Comput. 22,* 5, 1762–1771.

STRAZDINS, P. E. 1998. Transporting distributed BLAS to the Fujitsu AP3000 and VPP-300. In *Proceedings of the 8th Parallel Computing Workshop (PCW'98)*. 69–76.

WHALEY, R. C. AND DONGARRA, J. J. 1998. Automatically tuned linear algebra software. In *Proceedings of IEEE/ACM Conference on Supercomputing (SC'98)*.

WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. 2001. Automated empirical optimization of software and the ATLAS project. *Parall. Comput. 27,* 1–2, 3–35.

Received November 2005; revised November 2006, April 2007; accepted April 2007

High-Performance Implementation of the Level-3 BLAS

KAZUSHIGE GOTO
and
ROBERT VAN DE GEIJN
The University of Texas at Austin

A simple but highly effective approach for transforming high-performance implementations on cache-based architectures of matrix-matrix multiplication into implementations of other commonly used matrix-matrix computations (the level-3 BLAS) is presented. Exceptional performance is demonstrated on various architectures.

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

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Linear algebra, libraries, basic linear algebra subprograms, matrix-matrix operations

ACM Reference Format:

Goto, K. and van de Geijn, R. 2008. High-performance implementation of the level-3 BLAS. ACM Trans. Math. Softw., 35, 1, Article 4 (July 2008), 14 pages. DOI 10.1145/1377603.1377607 http://doi.acm.org/10.1145/1377603.1377607

4

1. INTRODUCTION

Attaining high performance for matrix-matrix operations such as symmetric matrix-matrix multiply (SYMM), symmetric rank-k update (SYRK), symmetric rank-2k update (SYR2K), triangular matrix-matrix multiply (TRMM), and triangular solve with multiple right-hand sides (TRSM) by casting the bulk of computation in terms of a general matrix-matrix multiply (GEMM) has become a generally accepted practice [Kågström et al. 1998]. Variants on this theme

This research was partially sponsored by NSF Grant CCF-0540926. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.
Authors' addresses: K. Goto, Texas Advanced Computing Center, The University of Texas at Austin, Austin, TX 78712; email: kgoto@tacc.utexas.edu; R. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: rvdg@cs.utexas.edu.

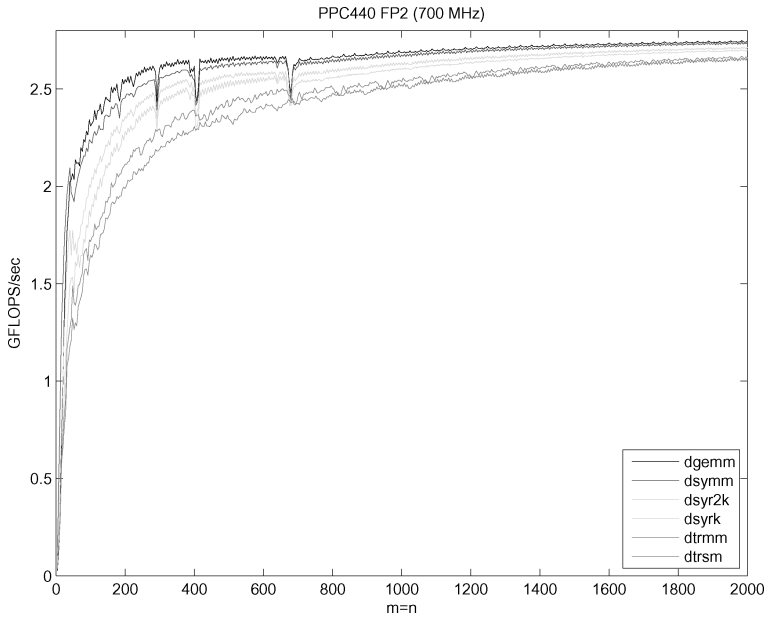


Fig. 1. Performance of all level-3 BLAS on the IBM PPC440 FP2 (700 MHz). (Curves in the figure appear, from top to bottom, in the order indicated in the legend.)

include loop-based algorithms and recursive algorithms, as well as hybrids that incorporate both of these [Elmroth et al. 2004]. In this article we show that better performance can be attained by specializing a high-performance GEMM kernel [Goto and van de Geijn 2008] so that it computes the desired operation. For the busy reader the results are previewed in Figure 1.

This article is organized as follows. In Section 2, we review the basic techniques behind a high-performance matrix-matrix multiplication implementation. More traditional techniques for implementing level-3 BLAS are reviewed in Section 3. Our alternative techniques are presented and used to obtain highly optimized implementations of SYMM, SYRK, SYR2K, TRMM, and TRSM, in Sections 4–7. Concluding remarks are given in the final section.

2. HIGH-PERFORMANCE IMPLEMENTATION OF MATRIX-MATRIX MULTIPLICATION

To understand how to convert a high-performance matrix-matrix multiplication (GEMM) implementation into a fast implementation for one of the other matrix-matrix operations that are part of the level-3 Basic Linear Algebra Subprograms (BLAS) [Dongarra et al. 1990], one has to first review the state-of-the-art of high-performance implementation of the GEMM operation. In this section we give a minimal description, referring the interested reader to Goto and van de Geijn [2008].

Consider the computation $C := AB + C$, where C, A, and B are $m \times n$, $m \times k$, and $k \times n$ matrices, respectively. Assume for simplicity that $m = b_m M$, $n = b_n N$, and $k = b_k K$, where M, N, K, b_m, b_n, and b_k are all integers. Partition as in

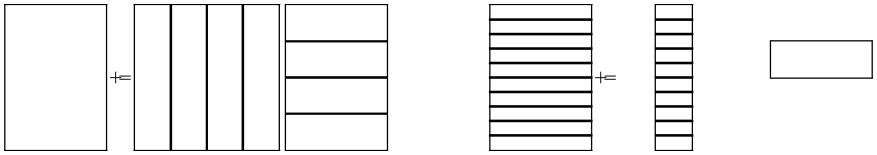


Fig. 2. Left: Partitioning of A and B. Right: Blocking for one individual panel-panel multiplication (GEPP) operation, $C := A_j B_j + C$.

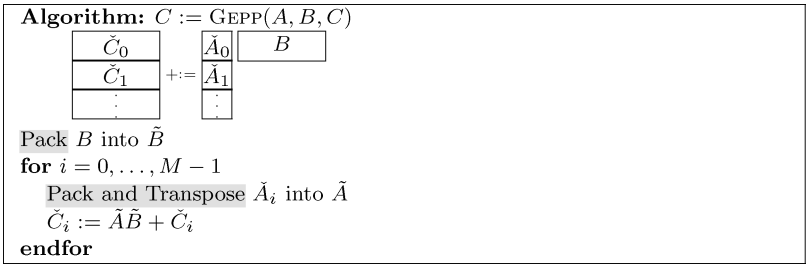


Fig. 3. Outline of optimized implementation of GEPP.

Figure 2(left):

$$A \rightarrow \left(A_0 \,\middle|\, A_1 \,\middle|\, \cdots \,\middle|\, A_{K-1} \right) \quad \text{and} \quad B \rightarrow \left(\begin{array}{c} \check{B}_0 \\ \hline \check{B}_1 \\ \hline \vdots \\ \hline \check{B}_{K-1} \end{array} \right),$$

where A_p and $\check{B}_p$ contain b_k columns and rows, respectively.[1] Then

$$C := A_0 \check{B}_0 + A_1 \check{B}_1 + \cdots + A_{K-1} \check{B}_{K-1} + C.$$

A typical high-performance implementation of GEMM will focus on making each update $C := A_p \check{B}_p + C$, which we will call a *panel-panel multiplication* (GEPP), as fast as possible. The overall performance of GEMM is essentially equal to that of each individual GEPP with panel width equal to an optimal size b_k.

Figure 3 gives a high-performance algorithm for the GEPP operation, $C := AB + C$, where the "k" dimension is b_k. The algorithm requires three highly optimized components:

—*Pack B:* A routine for packing B into a contiguous buffer. On some architectures this routine may also reorganize the data for specialized instructions used by the GEBP kernel routine described below.

—*Pack and transpose* $\check{A}_i$*:* A routine for packing $\check{A}_i$ into a contiguous buffer. Often this routine also transposes the matrix to improve the order in which it is accessed by the GEBP kernel routine.

—GEBP *kernel routine:* This routine computes $\check{C}_i := \tilde{A}\tilde{B} + \check{C}_i$ using the packed buffers. GEBP stands for General block-times-panel multiply.

[1]The ˇ is used to indicate a partitioning by rows in this paper.

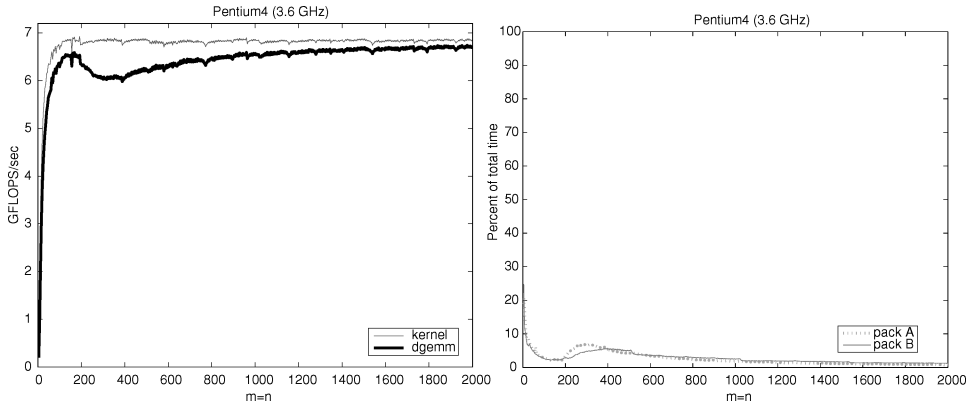


Fig. 4. Performance of GEMM on Pentium4 (3.6 GHz). Left: The performance of the GEBP kernel routine and GEMM is given by the curves labeled "Kernel" and"dgemm". Right: The curves labeled "Pack A" and "Pack B" indicate the percent of total time spent in each of these operations. In this graph $k = 192$.

On current architectures the size of $\check{A}_i$ is chosen to fill about half of the L2 cache (or the memory addressable by the TLB), as explained in Goto and van de Geijn [2008]. Considerable effort is required to tune each of these components, especially the GEBP kernel routine. In subsequent sections we will show how other level-3 BLAS can be implemented in such a way that this effort can be largely amortized by making minor modifications to the GEBP kernel.

In Figure 4 the performance and overhead of the various kernels is reported for the high-performance implementation of DGEMM (double-precision GEMM) from [Goto and van de Geijn 2008]. In that paper, the methodology is shown to be competitive with the DGEMM implementations of vendor libraries. It is this implementation upon which the remainder of this paper is based.

Throughout the article performance is presented for double precision (64-bit) computation of the target operation on a number of architectures:

Architecture	Clock (GHz)	Peak (GFLOPS/sec)	blocking size b_m	blocking size b_k	Vendor library
Pentium4 (R)	3.6	7.2	768	192	MKL 8.0.1
Itanium2 (R)	1.5	6	128	1024	MKL 8.0.1
Power 5	1.9	7.6	256	256	ESSL 4.2.0
PPC440 FP2	0.7	2.8	128	3072	not available to us

We also compare against ATLAS 3.7.11, a public-domain implementation of the BLAS [Whaley and Dongarra 1998], except for the Itanium2 system, on which ATLAS 3.7.8 attained better performance. In our graphs that report the rate of execution (GFLOPS/sec) the top line always represents the theoretical peak of the processor. The blocking sizes b_m and b_k are as indicated in the above table.

Key insights from Goto and van de Geijn [2008] are that (1) the submatrix $\hat{A}_i$ is typically non-square, (2) the cost of packing $\hat{A}_i$ is significant, which means that the column dimension of B should be large, and (3) the cost of packing B

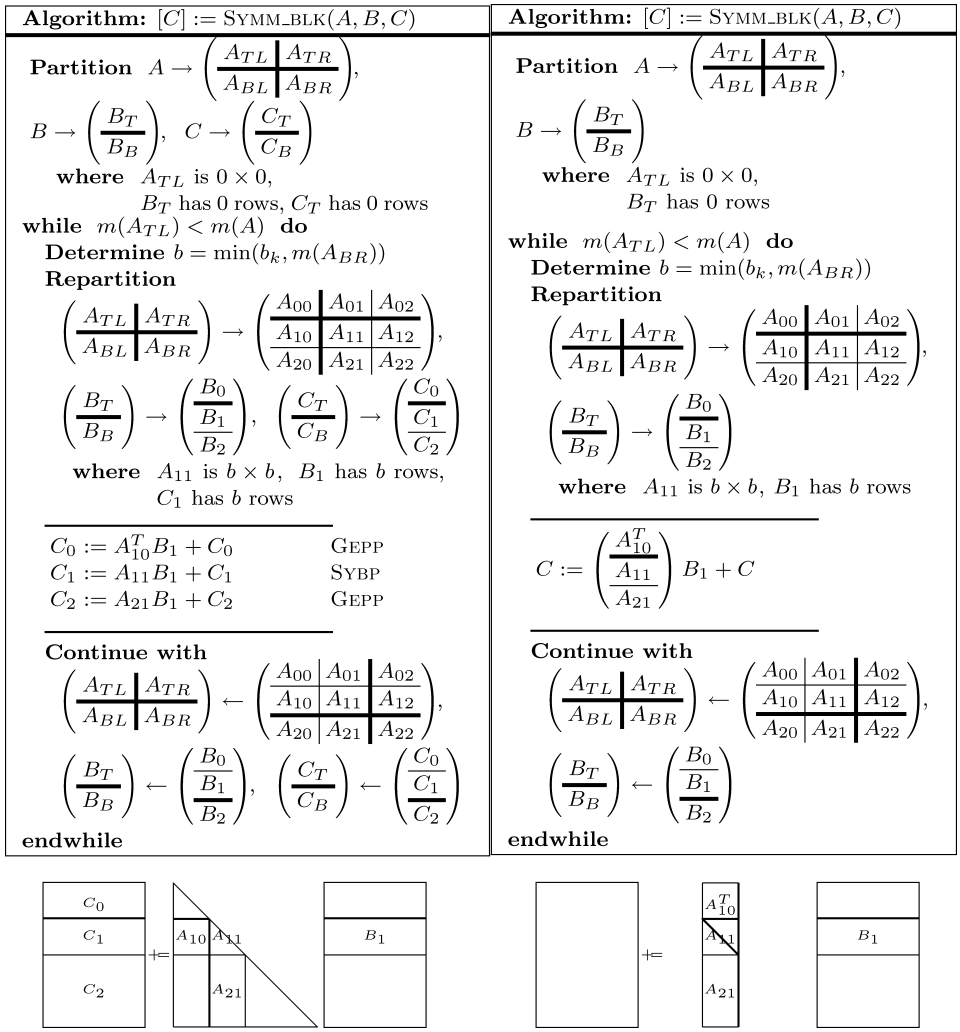


Fig. 5. Algorithms for computing SYMM. Left: Typical loop-based algorithm. Right: Casting in terms of a single GEPP.

is significant and should therefore be amortized over as many blocks of A as possible and repacking should be avoided.

3. TRADITIONAL APPROACHES FOR IMPLEMENTING THE LEVEL-3 BLAS

We use the symmetric matrix-matrix multiplication (SYMM), $C := AB+C$ where A is symmetric, as an example of how traditional approaches to implementing the Level-3 BLAS proceed. We will assume that only the lower triangular part of A is stored (in the lower triangular part of the array that stores A).

3.1 Loop-Based Approach

In Figure 5(left) we show a typical computation SYMM as a loop that traverses the matrices a block of rows and/or columns at a time. We believe the notation

used in that figure, which has been developed as part of the FLAME project, to be sufficiently intuitive not to require further explanation [Bientinesi and van de Geijn 2006; Gunnels et al. 2001]. Notice that the bulk of the computation is cast in terms of GEPP. The size of A_{11} in each iteration is chosen to equal the optimal b_k discussed in Section 2 so that the GEPP updates $A_{01}^T B_0$ and $A_{21}B_2$ attain near-optimal performance. Since typically b_k is small relative to m (the dimension of A) the update $C_1 := A_{11}B_1 + C_1$, which we will call a *symmetric block-matrix multiplication* (SYBM) requires relatively few operations. Letting n equal the column dimension of C, the total operation count of a SYMM operation is $2m^2n$ floating point operations (flops). A total of $2(m/b_k)b_k^2 n = 2mb_k n$ flops are in the SYBM computations and $2(m - b_k)mn$ in the GEPP operations. Even if the performance attained by the SYBM operations is less than that of a GEPP, the overall performance degrades only moderately if $m >> b_k$. In our case, SYBM is implemented by copying A_{11} into a temporary matrix, making it "general" by copying the lower triangular part to the upper triangular part, and calling GEMM.

There are two sources of complications and/or inefficiencies in this approach:

—The three operations ($C_0 := A_{10}^T B_1 + C_0$, $C_1 := A_{11}B_1 + C_1$, and $C_2 := A_{21}B_1 + C_2$) are typically treated as three totally separate operations, meaning that B_1 must be packed redundantly for each of the three operations. In Figure 4 it is shown that the packing of B_1 is a source of overhead for an individual GEPP operation that cannot be neglected.

—Since the shape and size of of C_1 and the $b_k \times b_k$ block of A_{11} in the SYBM operation do not match those of the corresponding submatrices in the GEPP operation, optimization of the SYBM operation is not a matter of making minor modifications to the kernel GEBP operation. A high-performance implementation would require a redesign of this kernel.

These problems are most noticeable when A is relatively small.

3.2 Recursive Algorithms

Partition

$$C \rightarrow \left(\frac{C_T}{C_B}\right), \quad A \rightarrow \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right), \quad \text{and} \quad B \rightarrow \left(\frac{B_T}{B_B}\right),$$

where A_{TL} is a $k \times k$ matrix and B_T and C_T are $k \times n$. Then $C := AB + C$ yields

$$\begin{aligned} \left(\frac{C_T}{C_B}\right) &= \left(\begin{array}{c|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right)\left(\frac{B_T}{B_B}\right) + \left(\frac{C_T}{C_B}\right) \\ &= \left(\frac{A_{TL}B_T + A_{BL}^T B_B + C_T}{A_{BL}B_T + A_{BR}B_B + C_B}\right). \end{aligned}$$

The terms $A_{TL}B_T + C_T$ and $A_{BR}B_B + C_B$ can be achieved by recursive calls to SYMM. This time the bulk of the computation is cast in terms of GEMM: $A_{BL}^T B_B$ and $A_{BL}B_T$. Typically the recursion stops when matrix A is relatively small, at which point A may be copied into a general matrix, after which GEMM can be employed for this small problem.

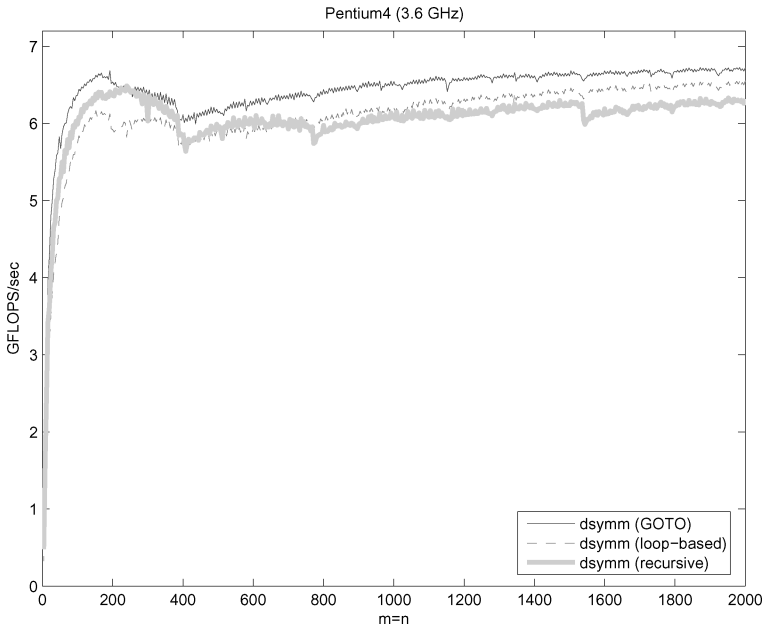


Fig. 6. Performance of different implementations of SYMM: The algorithm put forth later in this paper, a loop-based algorithm, and a recursive algorithm.

There are two sources of complications and/or inefficiencies in this recursive approach:

—The same panels of B will be packed multiple times as part of the individual calls to GEMM, which itself is cast in terms of GEPP operations.

—Unless care is taken the recursion will not create subproblems of sizes that are integer multiples of b_k, which causes the GEMM operations to attain less than optimal performance.

3.3 Performance

The performance of the two traditional approaches described in this section are reported in Figure 6. In that graph we also report the performance attained by the approach delineated later in this paper. The block size for the loop-based algorithm was taken to equal 192 while the recursion terminated when the size of the subproblem was less then or equal to 192.

4. SYMM

The alternative approach to implementing SYMM professed by this paper is simple to describe: Execute exactly the same algorithm as was employed for GEPP by modifying the routine that copies submatrices of matrix A into packed form to accommodate the symmetric nature of matrix A.

To understand this fully, first consider the algorithm in Figure 5(right). Notice that *if* A_{10}^T, A_{11}, and A_{21} are copied into a single panel of columns of width b_k, then the GEPP algorithm in Figure 3 can be executed. This approach is

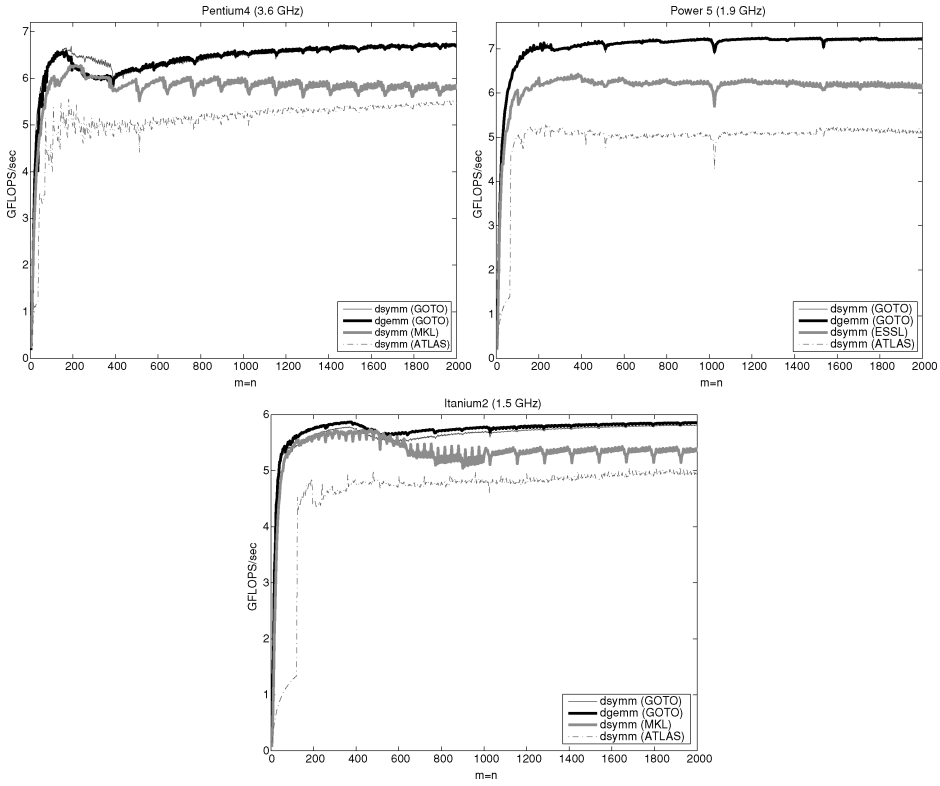
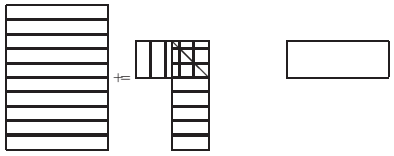


Fig. 7. Performance of SYMM relative to GEMM. (For the Power5 architecture the line for `dsymm` and `dgemm` are almost coincident. The `dsymm` line lies ever so slightly above that of `dgemm`.)

inefficient in the sense that these submatrices are first copied and then subsequently packed as part the GEPP algorithm. This suggests that instead of first copying the three parts into a single column panel, the GEPP algorithm should be modified so that the copying is done as needed, a block of $b_m \times b_k$ at a time, as illustrated by



While simple, the method has a number of immediate benefits:

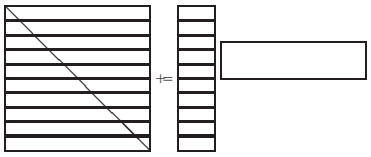
—The packing of the block of rows of B is amortized over all computation with A_{10}, A_{11}, and A_{21}.
—The routine for packing submatrices of A needs only be modified slightly.
—The exact same kernel GEBP routine as for implementing GEPP can be used.

Interestingly enough, the approach yields performance that often *exceeds* that of GEMM, as shown in Figure 7.

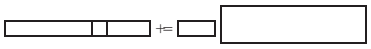
5. SYRK AND SYR2K

Next we discuss the symmetric rank-k (SYRK) and symmetric rank-2k (SYR2K) updates: $C := AA^T + C$ and $C := AB^T + BA^T + C$, where C is an $m \times m$ symmetric matrix and A and B are $m \times k$. We will assume that only the lower triangular part of C is stored.

Let us focus on SYRK first. As for the GEMM and SYMM operations it is important to understand how one panel-panel multiply is optimized: the case where A contains b_k columns ($k = b_k$) Mimicking the GEPP implementation yields



The idea now is that the computation of each row panel of C is modified to take advantage of the fact that only the part that lies at or below the diagonal needs to be updated. One straightforward way to accomplish this is to break each such row panel into three parts:



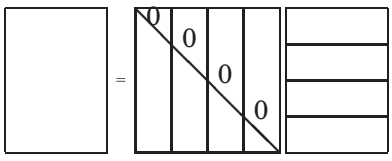
The kernel GEBP routine can be used to update the left part. A special kernel updates the lower triangular block on the diagonal, and the right part is not updated at all.

The implementation of SYR2K is a simple extension of this, where a slight optimization is that each row panel of C is updated with the appropriate part of both AB^T and BA^T since this keeps the panel of C in the L3 cache, if present. Again, the performance is impressive, as illustrated in Figure 8.

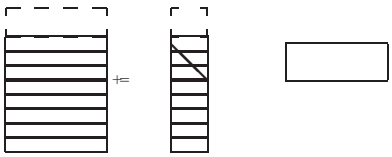
6. TRMM

We will examine the specific case of the triangular matrix-matrix multiply (TRMM) $B := LB$, where L is a lower triangular $m \times m$ matrix and B is $m \times n$.

Again, this operation can be cast in terms of a sequence of panel-panel multiplies:



An examination of how the GEPP algorithm can be modified for the special needs of TRMM yields



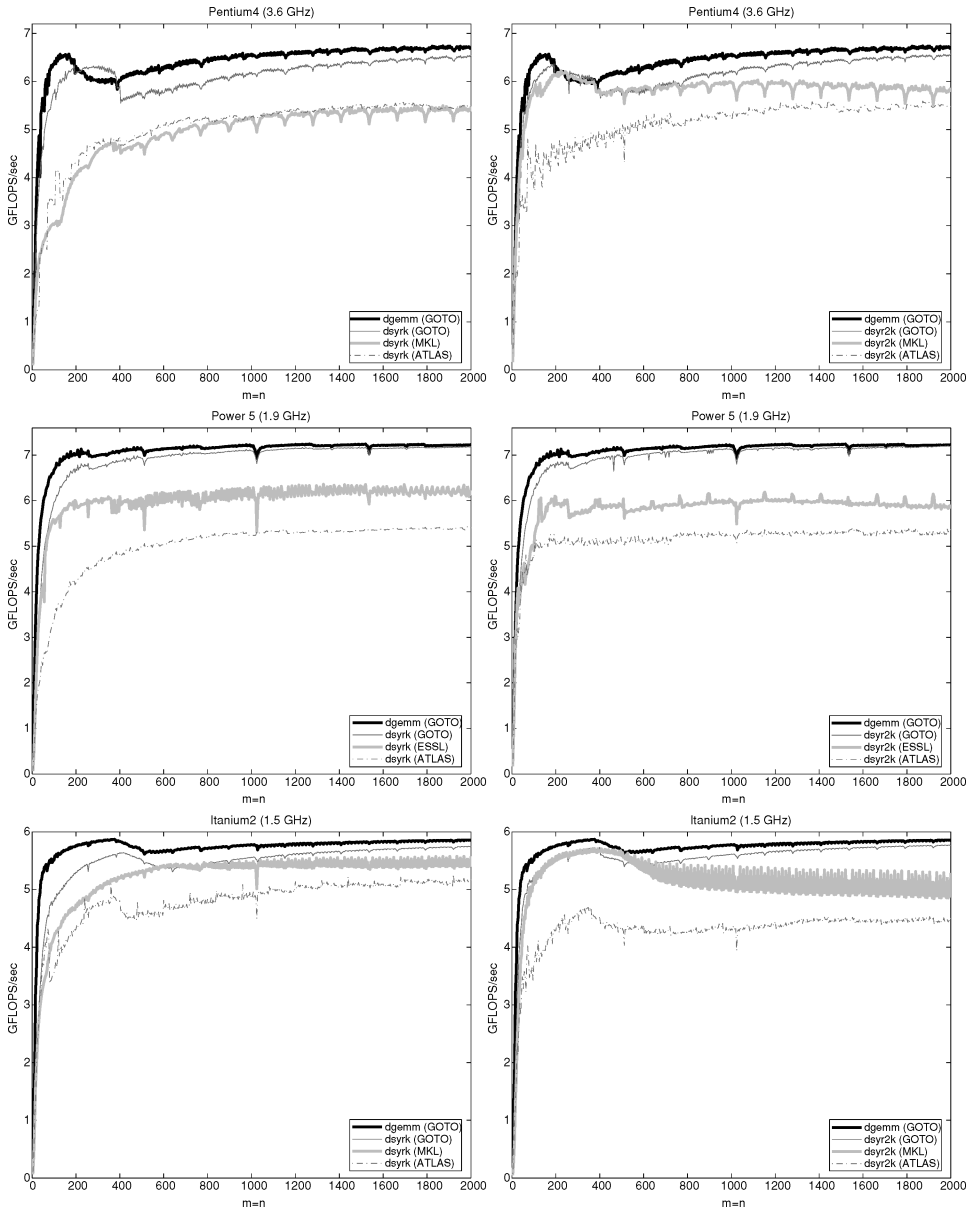


Fig. 8. Performance of SYRK (left) and SYR2K (right) relative to GEMM.

One notices that again most of the computation can be cast in terms of the kernel GEBP routine, except for the computation with the blocks that contain part of the diagonal. There are a number of ways of dealing with those special blocks:

—As the block is packed and transposed the elements in from the upper triangular part can be set to zero, after which the kernel GEBP routine can be used

without modification. The advantage is that only the packing routine needs to be modified slightly. The disadvantage is that considerable computation is performed with elements that equal zero.

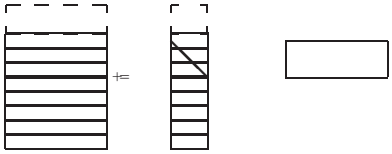
—Modify the kernel GEBP routine so that it does not compute with elements that lie above the diagonal. Conceptually this means changing a few loop-bounds. In practice there is loop-unrolling that is incorporated in the kernel GEBP routine that makes this somewhat more complex. One possibility for overcoming this while making only slight changes to the kernel routine is to set those elements that lie in a region covered by the loop-unrolling to zero and to compute with those but not other elements that lie above the diagonal. This can then be accomplished by only modifying loop-bounds in the kernel routine without disturbing code related to loop-unrolling.

We favor the second solution in our implementations. The performance of the TRMM routine is demonstrated in Figure 9(left).

7. TRSM

We will examine the specific case of the triangular solve with multiple right-hand sides (TRSM) $B := L^{-1}B$, where L is a lower triangular $m \times m$ matrix and B is $m \times n$. An algorithm that casts most computation in terms of GEPP is given in Figure 10.

Let us examine the combined updates $B_1 := L_{11}^{-1}B_1$ and $B_2 := B_2 - L_{21}B_1$:

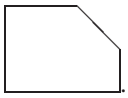


It is in how to deal with the blocks that contain the diagonal that complications occur. For these blocks

—B_1 will have been copied into a packed array $\tilde{B}$.

—The current row panel will have b_m rows. Let us denote this row panel by the matrix C.

—A typical block of L_{11}, A, that contains the diagonal will have the shape

$A =$ 

—Partitioning C, A, and $\tilde{B}$ as

$$C \rightarrow \begin{array}{|c|}\hline C_T \\ \hline C_B \\ \hline \end{array}, \quad A \rightarrow \begin{array}{|cc|}\hline A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \\ \hline \end{array}, \quad \text{and} \quad \tilde{B} \rightarrow \begin{array}{|c|}\hline \tilde{B}_T \\ \hline \tilde{B}_B \\ \hline \end{array}$$

the operation to be performed can then given by

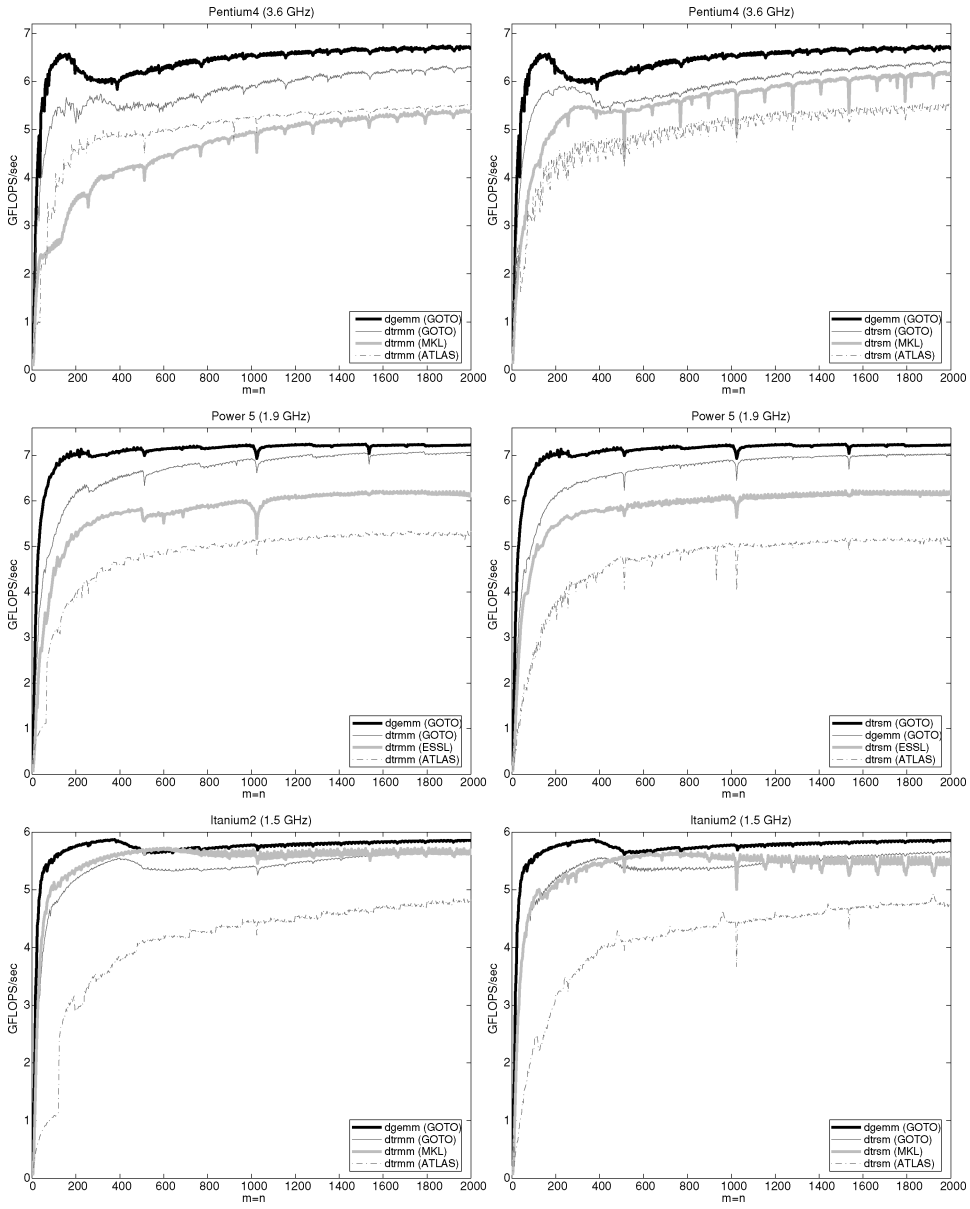


Fig. 9. Performance of TRMM (left) and TRSM (right) relative to GEMM.

—$C_T := A_{TR}^{-1}(C_B - A_{TL}C_T)$.
The data in C_T coincides with the part of B that was copied into $\tilde{B}$. Thus the result in C_T needs also be updated in the corresponding part of $\tilde{B}$.
—$C_B := C_B - A_{BL}\tilde{B}_T - A_{BR}\tilde{B}_B$.
Again, the data in C_B coincides with the part of B that was copied into $\tilde{B}$. Thus the result in C_B needs also be updated in the corresponding part of $\tilde{B}$.

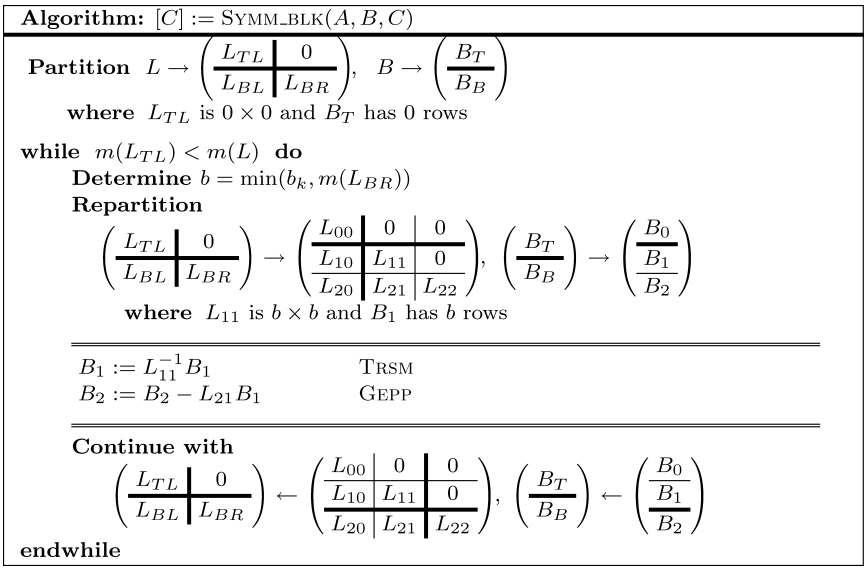


Fig. 10. Algorithm for computing TRSM that casts most computation in terms of GEPP.

Clearly, the kernel that implements this requires considerable care and cannot be simply derived from the kernel GEBP routine. Performance of our implementation is reported in Figure 9.

8. CONCLUSION

In this article, we have presented a simple yet highly effective approach to implementing level-3 BLAS routines by modifying the currently most effective technique for implementing matrix-matrix multiplication. The methodology inherently avoids unnecessary recopying of data into packed format. It suggests that routines like those that pack and kernel routines be exposed as building blocks for libraries.

The performance comparison with the MKL library on the Itanium2 architecture may appear to present a counterexample to the techniques advocated by this article, since for some operations the MKL implementation outperforms our implementations. We note that their implementations require substantially more effort than those supported by our work.

There are a number of other situations in which exposing these building blocks will become advantageous if not necessary.

—A typical LU factorization (with or without pivoting) performs a TRSM operation with a matrix that subsequently becomes an operand in a GEPP. This could allow a new packing of that data to be avoided if the packed array used in the implementation of the TRSM is saved. Similar redundant repacking of data is encountered in many LAPACK level routines by virtue of the strict layering of such libraries upon the BLAS interface.

—Implementation of level-3 BLAS on SMP or multi-core platforms could easily incur redundant packing operations by the different threads. Exposing the building blocks could avoid this, improving performance considerably. This is discussed for DGEMM in [Marker et al. 2007].

We believe this suggests that the standardization of interfaces to such building blocks is in order.

Implementations of the described techniques are available for essentially all current architectures. Libraries can be obtained from `http://www.tacc.utexas.edu/resources/software/`.

ACKNOWLEDGMENTS

The Itanium2 server used in this research was a donation by Hewlett-Packard. Access to the IBM PPC440 FP2 was arranged by Lawrence Livermore National Laboratory. The Texas Advanced Computer Center provided access to the other architectures. We would like to thank Dr. John Gunnels, members of the FLAME team, and the anonymous referees for their comments on an earlier draft of this paper.

REFERENCES

BIENTINESI, P. AND VAN DE GEIJN, R. 2006. Representing dense linear algebra algorithms: A farewell to indices. FLAME Working Note #17 TR-2006-10, Department of Computer Sciences, The University of Texas at Austin.

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

ELMROTH, E., GUSTAVSON, F., JONSSON, I., AND KÅGSTRÖM, B. 2004. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Rev. 46,* 1, 3–45.

GOTO, K. AND VAN DE GEIJN, R. A. 2008. Anatomy of a high-performance matrix multiplication. *ACM Trans. Math. Softw. 34,* 3.

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

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

MARKER, B., VAN ZEE, F. G., GOTO, K., QUINTANA-ORTÍ, G., AND VAN DE GEIJN, R. A. 2007. Toward scalable matrix multiply on multithreaded architectures. In *Proceedings of the International Euro-Par Conference*. A.-M. Kermarrec, L. Bougé, and T. Priol, Eds. Lecture Notes on Computer Science, vol. 4641. 748–757.

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

Received May 2006; revised April 2007; accepted October 2007

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

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

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

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

General Terms: Algorithms, Performance

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

ACM Reference Format:

Bientenesi, P., Gunter, B., and van de Geijn, R. A. 2008. Families of algorithms related to the inversion of a symmetric positive definite matrix. ACM Trans. Math. Softw. 35, 1, Article 3 (July 2008), 22 pages DOI 10.1145/1377603.1377606 http://doi.acm.org/10.1145/1377603.1377606

P. Bientinesi was with the University of Texas at Austin when this work was done.
This research was partially sponsored by NSF grants CCF-0342369 and ACI-0305163. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.
Authors' addresses: P. Bientinesi, Department of Computer Science, RWTH Aachen University, 52056 Aachen, Germany; email: pauldj@aices.rwth-aachen.de; B. Gunter, Delft University of Technology, Department of Earth Observation and Space Systems, 2629 HS, Delft, The Netherlands; email: b.c.gunter@tudelft.nl; R. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: rvdg@cs.utexas.edu.

1. INTRODUCTION

We discuss the need for the inclusion of families of algorithms and implementations in dense linear algebra libraries in order to more effectively address situation specific requirements. Special situations may be due to architectural features of a target platform or to application requirements. While this observation is not new, the general consensus in the community has been that traditional libraries are already too complex to develop when only one or two algorithms are supported, making it impractical to consider including all algorithms for all operations [Demmel and Dongarra 2005]. Obstacles include the effort required to identify candidate algorithms, backward compatibility with traditional development techniques, establishing the formal correctness[1] through extensive testing, numerical stability analyses,[2] and the time required for empirical tuning. Recent developments towards the systematic and mechanical development of libraries, as part of the Formal Linear Algebra Methods Environment (FLAME) project, suggest that many of these obstacles can be overcome if new software engineering approaches and tools are embraced. A departure from traditional development methods has the potential for greatly reducing the effort and expense of upgrading libraries as new architectures become available and new situations arise.

The FLAME project encompasses a large number of theoretical and practical tools. At the core is a new notation for expressing dense linear algebra algorithms [Quintana et al. 2001; Bientinesi and van de Geijn 2006]. This notation has a number of attractive features: (1) it avoids the intricate indexing into the arrays that store the matrices that often obscures the algorithm; (2) it raises the level of abstraction at which the algorithm is represented; (3) it allows different algorithms for the same operation and similar algorithms for different operations to be easily compared and contrasted; and (4) it allows the state of the matrix at the beginning and end of each loop (the loop-invariant)[3] to be concisely expressed. The notation supports a step-by-step process for deriving formally correct families of loop-based algorithms requiring as input only a mathematical specification of the operation [Bientinesi et al. 2005a]. As part of the project, Application Program Interfaces (APIs) for representing algorithms in code have been defined for a number of programming languages [Bientinesi et al. 2005b]. These APIs allow the code to closely resemble the formally correct algorithms so that (1) the implementation requires little effort and (2) the formal correctness of the algorithms implies the formal correctness of the implementations. The methodology is sufficiently systematic that it has been made mechanical using Mathematica [Bientinesi 2006]. The project is also working towards making numerical stability analysis [Bientinesi 2006] and performance analysis similarly systematic and mechanical [Gunnels 2001].

[1]Formal correctness in computer science refers to correctness in the absence of round-off errors.

[2]In the presence of finite precision arithmetic a numerically stable algorithm will yield an answer that equals the exact answer to a nearby problem.

[3]A formal definition of "loop-invariant" can be found in the book *A logical Approach to Discrete Math* [Gries and Schneider 1992]. We caution that this term is often used by the compiler community with a different (almost opposite) meaning.

The breadth of the methodology has been shown to include all of the Basic Linear Algebra Subprograms (BLAS) [Dongarra et al. 1988, 1990], many of the higher level dense linear algebra operations supported by the Linear Algebra Package (LAPACK) [Anderson et al. 1999] and all the operations in the RECSY library [Jonsson and Kågström 2002a, 200b]. The primary contribution of this paper is to highlight that multiple algorithms for a single operation must be supported by a complete library. This way the user, or, better yet, an expert system can select the best performing or the most appropriate algorithm for a given situation.

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

The inverse of a SPD matrix A is typically obtained by first computing the upper triangular Cholesky factor R of A, $A = R^T R$, after which $A^{-1} = (R^T R)^{-1} = R^{-1} R^{-T}$ can be computed by first inverting the matrix R ($U = R^{-1}$) and then multiplying the result by its transpose ($A^{-1} = UU^T$). We will show that there are multiple loop-based algorithms for each of these three operations, all of which can be orchestrated so that the result overwrites the input without requiring temporary space. Also presented will be two algorithms that overwrite A by its inverse without the explicit computation of these intermediate results, requiring only a single sweep through the matrix, as was already briefly mentioned in Quintana et al. [2001]. The performance benefit of the single-sweep algorithm for a distributed memory architecture is illustrated in Figure 1 in which the best algorithm for inverting a SPD matrix proposed in this paper and implemented with PLAPACK [van de Geijn 1997] is compared with the current three-sweep algorithm supported by ScaLAPACK [Choi et al. 1992]. A secondary contribution of this paper lies with the thorough treatment of loop-based algorithms, implementations, and performance for these operations. Many references to classic inversion methods can be found in Householder [1964] and Higham [2002]. An in-place procedure for inverting positive definite matrices by the Gauss-Jordan method is given by Bauer and Reinsch [1970]. Recent advances in data formats are applied to matrix inversion in Andersen et al. [2002] and Georgieva et al. [2000].

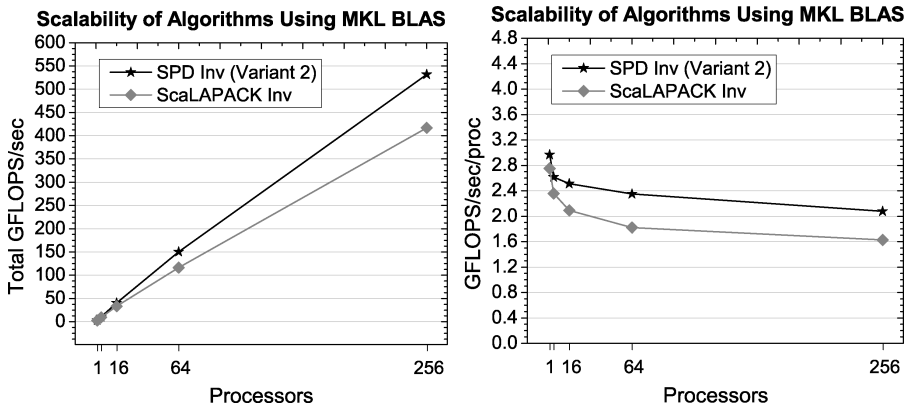


Fig. 1. Comparison of the performance of the ScaLAPACK routine for inverting a SPD matrix and our best one-sweep algorithm implemented with PLAPACK, as a function of the number of processing nodes, p. The problem size is chosen to equal $5000\sqrt{p}$ so that the memory use per processing node is constant.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

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

2. ALGORITHMS FOR THE INDIVIDUAL SWEEPS

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

2.1 Cholesky Factorization: $A := \text{CHOL}(A)$

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

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

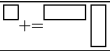
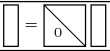
Name	Operation	Shape	Used in			
			Chol	R^{-1}	UU^T	A^{-1}
DOT	Dot product		1,2		2,3	1
SCAL	Scaling of vector		2,3	1,2,3	1,2	1,2
GEMV	General matrix-vector multiply		2		2	
SYMV	Symmetric matrix-vector multiply					1
GER	General rank-1 update			3		2
SYR	Symmetric rank-1 update		3		1	1,2
TRMV	Triangular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

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

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

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

where the base case for a 1×1 matrix $A = \alpha$ is $\text{CHOL}(A) = \sqrt{\alpha}$. In this definition, $\hat{A}$ represents the original contents of A,[4] the quadrants A_{TL}, R_{TL}, and $\hat{A}_{TL}$ are all square and of equal dimensions, and $\star$ indicates the symmetric part of the matrix that is not stored. It is this recursive definition, the Partitioned Matrix Expression (PME) in FLAME terminology, that is the input to the FLAME methodology for generating loop-based algorithms.

Three pairs of algorithmic variants for computing the Cholesky factorization are presented in Figure 4. The function $m(X)$ returns the number or rows of matrix X. For details on the notation used [Bientinesi et al. 2005a, 2005b]; Bientinesi and van de Geijn [2006]. The algorithms on the left are *unblocked* algorithms, meaning that each iteration moves the computation along by one row and column. Casting the algorithm in terms of matrix-matrix operations (level-3 BLAS [Dongarra et al. 1990]) allows high performance to be attained [Dongarra et al. 1991]. This is achieved by the *blocked* algorithms on the right, which move through the matrix by blocks of b rows and columns.

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

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

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

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

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

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

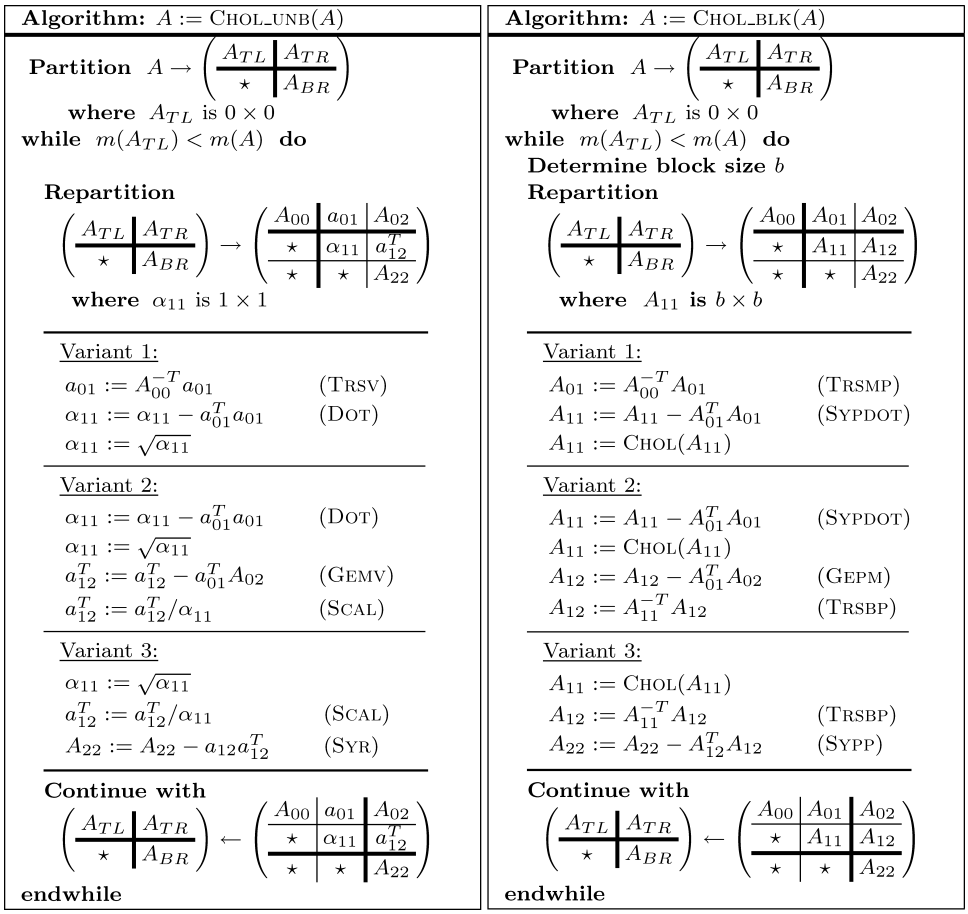


Fig. 4. Unblocked and blocked algorithms for computing the Cholesky factorization. We indicate within parenthesis the name of the operation being performed. A complete list of basic operations, with emphasis on the shape of the operands is given in Figure 2.

space is required. The PME for this operation is

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

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

Three blocked algorithms are given in Figure 5(left). They, respectively, maintain the loop-invariants in Figure 6(left). (Note again how the loop-invariants relate to the PME.) For each blocked algorithm there is a corresponding unblocked algorithm, which is not presented. Also, three more pairs of unblocked and blocked algorithms exist that sweep through the matrix from the bottom-right to the top-left. Finally, two more blocked and unblocked pairs of algorithms that are correct in the absense of round-off error but numerical unstable can

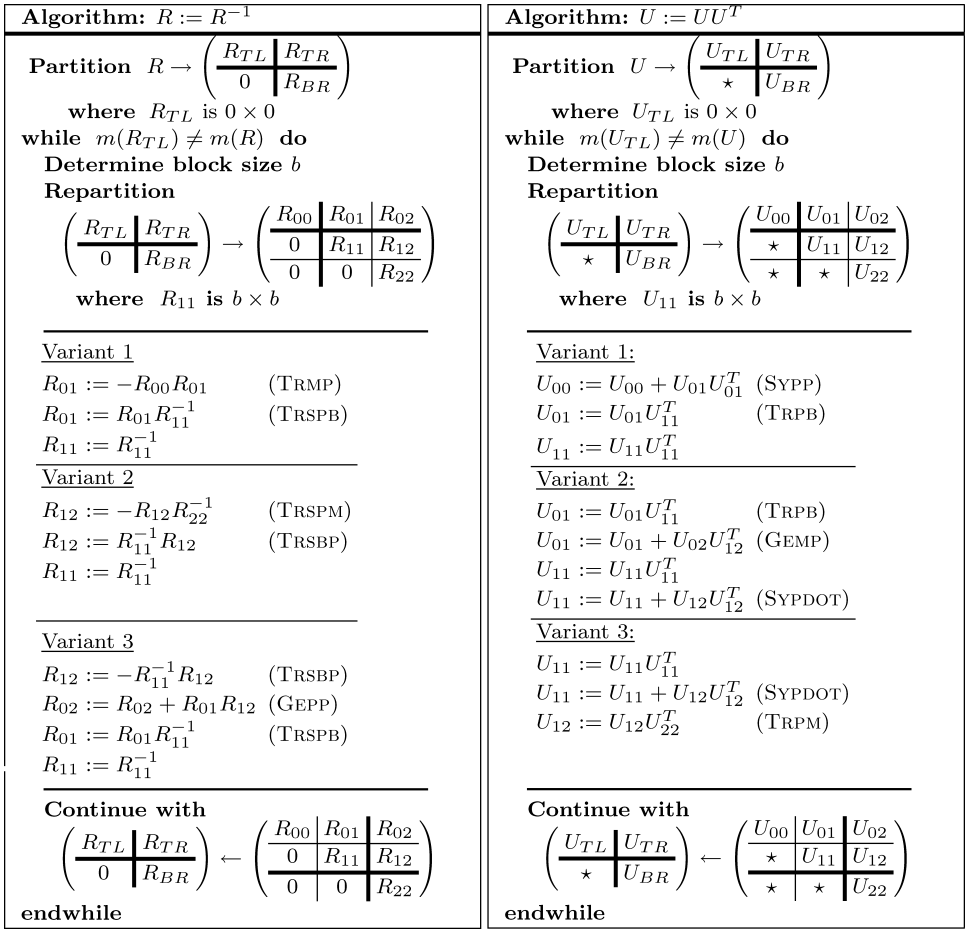


Fig. 5. Blocked algorithms for inverting a triangular matrix and for multiplying a triangular matrix by its transpose.

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

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

be derived. We will only consider the three numerically stable algorithms in Figure 5(left) [Bientinesi et al. 2006; Higham 2002].

2.3 Triangular Matrix Multiplication by Its Transpose: $C = UU^T$

The PME for this operation is

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

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

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

2.4 Three-Sweep Algorithms

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

3. ONE-SWEEP ALGORITHMS

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

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

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

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

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

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

Variant 1:	Variant 2:
$\left(\begin{array}{c\|c} \hat{A}_{TL}^{-1} & \hat{A}_{TR} \\ \hline \star & \hat{A}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} \hat{A}_{TL}^{-1} & -\hat{A}_{TL}^{-1}\hat{A}_{TR} \\ \hline \star & \hat{A}_{BR} - \hat{A}_{TR}^{T}\hat{A}_{TL}^{-1}\hat{A}_{TR} \end{array}\right)$

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

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

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

$Aux := -A_{00}A_{01}$ (SYMP)
$A_{11} := A_{11} + A_{01}^{T}Aux$ (GEPDOT)
$A_{11} := \text{CHOL}(A_{11})$
$Aux := Aux\, A_{11}^{-1}$ (TRSPB)
$A_{01} := Aux\, A_{11}^{-T}$ (TRSPB)
$A_{00} := A_{00} + Aux\, Aux^{T}$ (SYPP)
$A_{11} := A_{11}^{-1}$
$A_{11} := A_{11}A_{11}^{T}$

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

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

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

$A_{11} := \text{CHOL}(A_{11})$
$A_{01} := A_{01}A_{11}^{-1}$ (TRSPB)
$A_{00} := A_{00} + A_{01}A_{01}^{T}$ (SYPP)
$A_{12} := A_{11}^{-T}A_{12}$ (TRSBP)
$A_{02} := A_{02} - A_{01}A_{12}$ (GEPP)
$A_{22} := A_{22} - A_{12}^{T}A_{12}$ (SYPP)
$A_{01} := A_{01}A_{11}^{-T}$ (TRSPB)
$A_{12} := -A_{11}^{-1}A_{12}$ (TRSBP)
$A_{11} := A_{11}^{-1}$
$A_{11} := A_{11}A_{11}^{T}$

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

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

considering the alternative PME

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

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

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

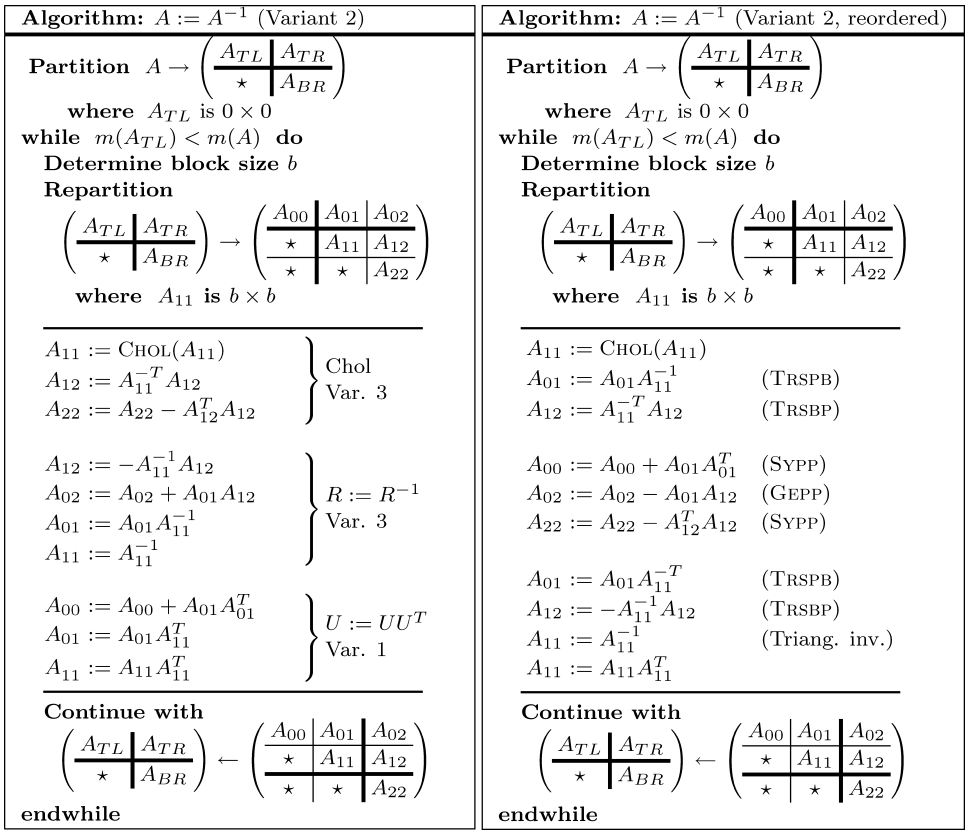


Fig. 9. One-sweep algorithm for inverting a SPD matrix as a merging of three sweeps. As a side effect of the reordering of the updates, the sign of the operands in the right column might be the opposite with respect to the corresponding update in the left column.

The real benefit of the one-sweep algorithm in Figure 9(left) comes from the following observation: The order of the updates in that variant can be changed as in Figure 9(right), so that the most time consuming computations ($A_{22} - A_{12}^T A_{12}$, $A_{00} + A_{01} A_{01}^T$, and $A_{02} + A_{01} A_{12}$) can be scheduled to be computed simultaneously:

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

On a distributed memory architecture, where the matrix is physically distributed among memories, there is the opportunity to: 1) consolidate the communication among processors by first performing the collective communications for the three updates followed by the actual computations, and 2) improve

load-balance since during every iteration of the merged algorithm, on each element of the quadrants A_{00}, A_{02} and A_{22} the same amount of computation is performed.

4. DIFFERENT ALGORITHMS FOR DIFFERENT SITUATIONS

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

4.1 Performance

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

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

Blocked algorithms cast most computation in terms of one of the matrix-matrix multiplies (GEPP, GEMP, GEPM, SYPP, etc.) [Dongarra et al. 1990; Kågström et al. 1998; Anderson et al. 1999]. There are architectural reasons why the rank-k updates (GEPP and SYPP) on current sequential architectures inherently attain somewhat better performance than the other cases [Goto and van de Geijn 2002; Gunnels et al. 2001a]. As a result, it is typically best to pick the algorithmic variant that casts most computation in terms of those cases of matrix-matrix multiplication. (Note that this means a different algorithmic variant is preferred than was for the corresponding unblocked algorithm.)

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

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

We have thus reasoned how performance depends not just on what operation is performed, but even on the shape of the operands that are involved in the operation. A taxonomy of operations that exposes the shape of the operands is given in Figure 2. The algorithms that were presented earlier in this paper

were annotated to expose the operations being performed and the legends of the graphs in Section 5 indicate the operation in which most computation is cast, using this taxonomy.

4.2 Algorithmic Fault-Tolerance

There is a real concern that some future architectures will require algorithmic fault-tolerance to be a part of codes that execute on them. There are many reasons quoted, including the need for low power consumption, feature size, and the need to use off-the-shelf processors in space where they are subjected to cosmic radiation [Gunnels et al. 2001b]. Check-pointing for easy restart partially into the computation is most easily added to an algorithm that maintains a loop-invariant where each quadrant is either completely updated or not updated at all. For such algorithms it is easy to keep track of how far into the matrix the computation has progressed.

4.3 Related Operations

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

4.4 Impact on Related Computer Science Research and Development

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

5. PERFORMANCE EXPERIMENTS

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

5.1 Implementations

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

they hide the indexing that makes coding in a traditional style error-prone and time-consuming.

The FLAME/C (C) and PLAPACK (C interfaced with MPI) APIs [Bientinesi et al. 2005b; van de Geijn 1997; Chtchelkanova et al. 1997; Gropp et al. 1994; Snir et al. 1996] were used for all the implementations, making the coding effort manageable.

5.2 Platforms

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

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

On this architecture, we compared performance when parallelism was attained in two different ways: 1) Implementing the algorithms with PLAPACK, which employs message passing via calls to IBM's MPI library; and 2) invoking the sequential FLAME/C implementations with calls to the multithreaded BLAS that are part of IBM's ESSL library as well as the GotoBLAS [Goto and van de Geijn 2008].

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

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

5.3 Data Distribution

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

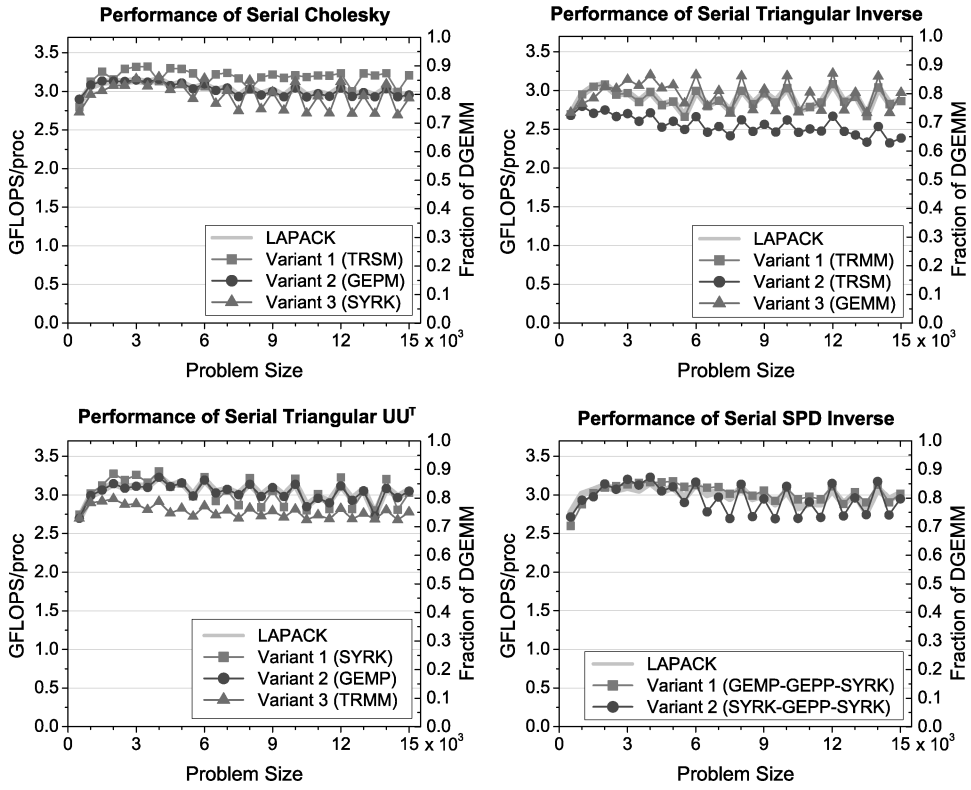


Fig. 10. Sequential performance on the IBM Power4 system.

5.4 Reading the Graphs

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

5.5 Sequential Performance

In Figure 10 we show performance on a single CPU of the IBM Power 4 system. In these experiments, a block size of 96 was used for all algorithms. From the graphs, it is obvious which algorithmic variant was incorporated in LAPACK.

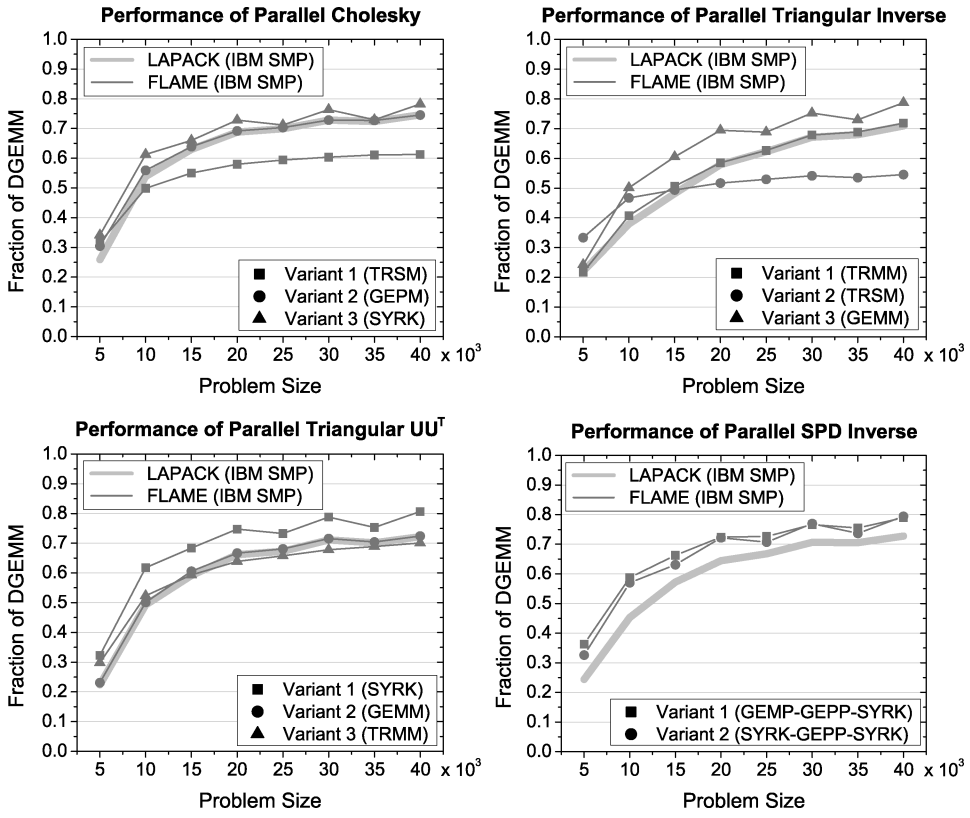


Fig. 11. Parallel performance on the 16 CPU IBM Power 4 SMP system.

5.6 Parallel performance

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

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

For the Cholesky factorization, the PLAPACK Variant 1 performs substantially worse than the other variants. This is due to the fact that this variant is rich in triangular solves with a limited number of right-hand sides. This operation inherently does not parallelize well on distributed memory architectures due to dependencies. Interestingly, Variant 1 for the Cholesky factorization attains the best performance in the sequential experiment on the same machine.

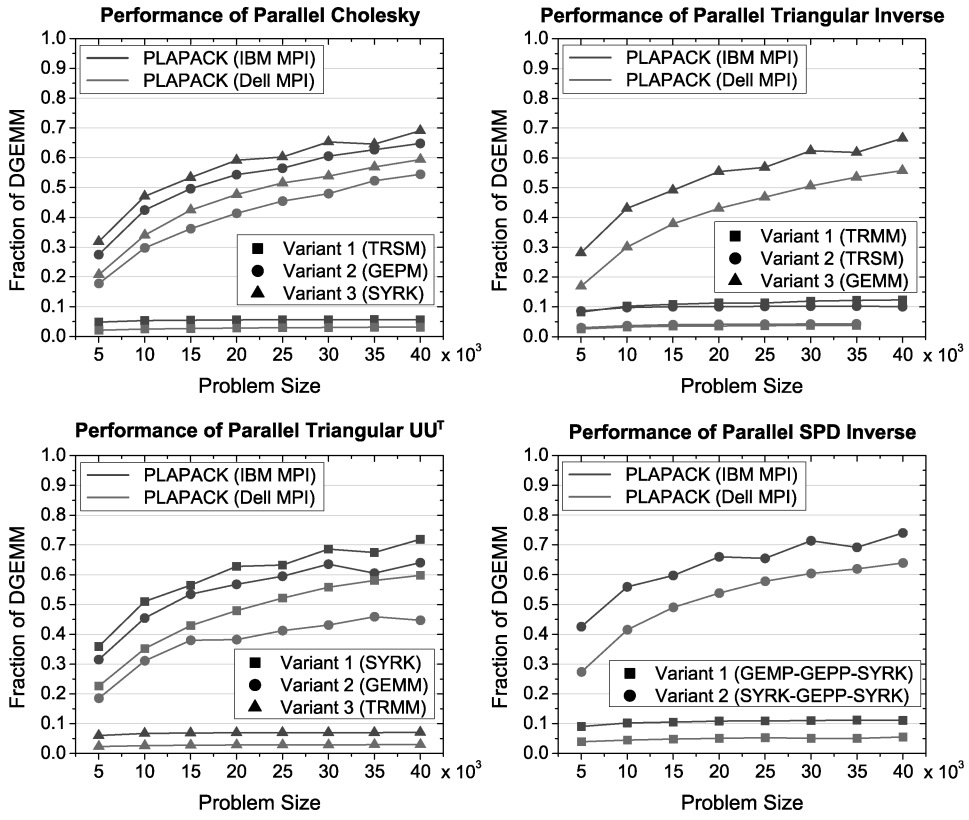


Fig. 12. Parallel performance for PLAPACK-based implementations (C interfaced with MPI).

The PLAPACK implementations of Variants 1 and 2 for computing R^{-1} do not perform well. Variant 1 is rich in triangular matrix times panel-of-columns multiply where the matrix being multiplied has a limited number of columns. It is not inherent that this operation does not parallelize well. Rather, it is the PLAPACK implementation for that BLAS operation that is not completely optimized. Similar comments apply to PLAPACK Variant 3 for computing UU^T and PLAPACK Variant 1 for computing the inversion of a SPD matrix. This shows that a deficiency in the performance of a specific routine in a parallel BLAS library (provided by PLAPACK in this case) can be overcome by selecting an algorithmic variant that casts most computation in terms of a BLAS operation that does attain high performance.[7] Note the cross-over between the curves for the SMP Variants 2 and 3 for the parallel triangular inverse operation. This shows that different algorithmic variants may be appropriate for different problem sizes.

It is again obvious from the graphs which algorithmic variant is used for each of the three sweeps as part of LAPACK. The LAPACK curve does not match

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

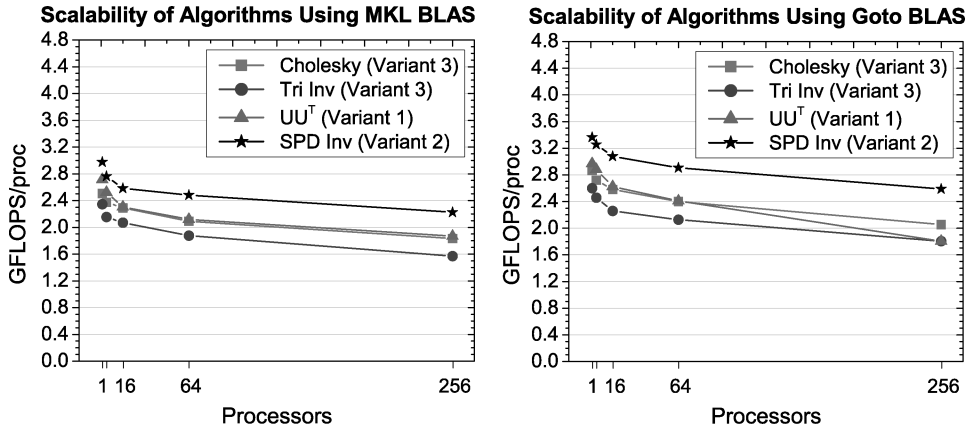


Fig. 13. Scalability on the Dell Cluster. Here the matrix size is scaled to equal $5000 \times \sqrt{p}$ so that memory use per processor is held constant. Left: when linked to MKL 7.X. Right: when linked to GotoBLAS 0.97.

either of the FLAME variants in the SPD inversion graph since LAPACK uses a three-sweep algorithm.

5.7 Scalability

In Figure 13 we report the scalability of the best algorithmic variants for each of the four operations when executed on the Cray-Dell cluster. It is well-known that for these types of distributed memory algorithms it is necessary to scale the problem size with the square root of the number of processors, so that memory-use per processor is kept constant [Hendrickson and Womble 1994; Stewart 1990]. Notice that as the number of processors is increased, the performance per node that is attained eventually decreases very slowly, indicating that the implementations are essentially scalable.

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

We examined the benefits of consolidating the collective communications and improving the load balancing in the single-sweep algorithm. In Figure 14(left) we show improvements in raw performance on the Cray-Dell system. The improvement over three-sweep algorithm is quite substantial, in the 15–30% range. Figure 14(right) shows the time savings gained for the PLAPACK implementations of the SPD inverse algorithms.

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

5.9 Comparison with ScaLAPACK

In Figure 1 we had already shown a performance comparison with ScaLAPACK on the Dell-Cray cluster. It verifies that our implementations rival and even surpass those of a library that is generally considered to be of high quality

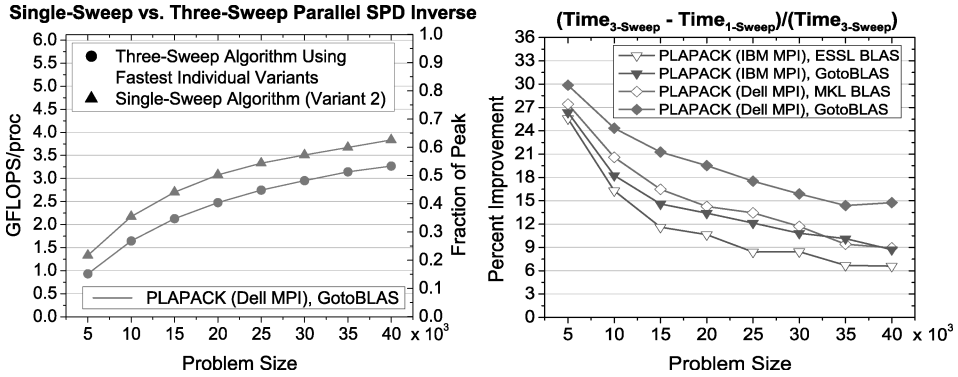


Fig. 14. Comparison of the Three-Sweep and Single-Sweep SPD inverse algorithms. The left panel shows the performance difference for the case run on the Cray-Dell system linked to the GotoBLAS. The right panel shows the wall-clock savings for all PLAPACK cases.

and scalable. ScaLAPACK requires the nodes to be logically viewed as an $r \times c$ mesh and uses a block-cyclic distribution of matrices to the nodes. For the ScaLAPACK experiments we determined that $r = c$ attained the best performance and a block size of 32 or 64 was used depending on which achieved better performance. (A block size of 128 achieved inferior performance since it affected load balance.) For the PLAPACK experiments $r = c$, a distribution block size of 32 and an algorithmic block size of 96 was used.

6. CONCLUSION

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

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

The article raises many new questions. In particular, the availability of many algorithms and implementations means that a decision must be made as to when to use what algorithm. One approach is to use empirical data from performance experiments to tune the decision process. This is an approach that

has been applied in the simpler arena of matrix-matrix multiplication (DGEMM) by the PHiPAC and ATLAS projects [Bilmes et al. 1997; Whaley and Dongarra 1998]. An alternative approach would be to carefully design every layer of a library so that its performance can be accurately modeled [Dackland and Kågström 1996]. We intend to pursue a combination of these two approaches.

ACKNOWLEDGMENTS

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

More Information

For more information on FLAME and PLAPACK visit

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

REFERENCES

ANDERSEN, B. S., GUNNELS, J. A., GUSTAVSON, F. G., AND WASNIEWSKI, J. 2002. A recursive formulation of the inversion of symmetric positive definite matrices in packed storage data format. In *Proceedings of the Applied Parallel Computing New Paradigms for HPC in Industry and Academia 7th International Workshop (PARA'02)* . Lecture Notes in Computer Science, vol. 2367. Springer, 287–296.

ANDERSON, E., BAI, Z., BISCHOF, C. H., BLACKFORD, S., DEMMEL, J. W., DONGARRA, J. J., CROZ, J. J. D., GREENBAUM, A., HAMMARLING, S. J., MCKENNEY, A., AND SORENSEN, D. C. 1999. *LAPACK Users' Guide*, 3rd Ed. Society for Industrial Mathematics.

BAUER, F. L. AND REINSCH, C. 1970. Inversion of positive definite matrices by the Gauss-Jordan methods. In *Handbook for Automatic Computation Vol. 2: Linear Algebra*, J. H. Wilkinson and C. Reinsch, Eds. Springer, Berlin, Germany, 45–49.

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

BIENTINESI, P., GUNNELS, J. A., MYERS, M. E., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2005a. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Softw. 31,* 1, 1–26.

BIENTINESI, P., GUNTER, B., AND VAN DE GEIJN, R. 2006. Families of algorithms related to the inversion of a symmetric positive definite matrix. FLAME Working Note #19 CS-TR-06-20, Department of Computer Sciences, The University of Texas at Austin.

BIENTINESI, P., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2005b. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Trans. Math. Softw. 31,* 1.

BIENTINESI, P. AND VAN DE GEIJN, R. 2006. Representing dense linear algebra algorithms: A farewell to indices. FLAME Working Note #17 CS-TR-06-10, Department of Computer Sciences, The University of Texas at Austin.

BILMES, J., ASANOVIĆ, K., WHYE CHIN, C., AND DEMMEL, J. 1997. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*. Vienna, Austria.

BLACKFORD, L. S., CHOI, J., CLEARY, A., D'AZEVEDO, E., DEMMEL, J., DHILLON, I., DONGARRA, J., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D., AND WHALEY, R. C. 1997. *ScaLAPACK Users' Guide*. SIAM.

CHOI, J., DONGARRA, J., POZO, R., AND WALKER, D. 1992. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. In *Proceedings of the 4th Symposium*

on the Frontiers of Massively Parallel Computation. IEEE Computer Society Press, 120–127.

CHTCHELKANOVA, A., GUNNELS, J., MORROW, G., OVERFELT, J., AND VAN DE GEIJN, R. 1997. Parallel implementation of BLAS: General techniques for level 3 BLAS. *Concurrency: Prac. Exper. 9,* 9, 837–857.

DACKLAND, K. AND KÅGSTRÖM, B. 1996. A hierarchical approach for performance analysis of scaLAPACK-based routines using the distributed linear algebra machine. In *Proceedings of the 3rd International Workshop on Applied Parallel Computing, Industrial Computation and Optimization (PARA'96)*. Lecture Notes in Computer Science, vol. 1184. Springer-Verlag, 186–195.

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

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

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

DONGARRA, J., DUFF, I., SORENSEN, D., AND VAN DER VORST, H. A. 1991. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA.

GEORGIEVA, G., GUSTAVSON, F. G., AND YALAMOV, P. Y. 2000. Inversion of symmetric matrices in a new block packed storage. In *Proceedings of the 2nd International Conference on Numerical Analysis and Its Applications (NAA'00)*. Springer, 333–340.

GOTO, K. AND VAN DE GEIJN, R. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw*. *34*, 3.

GOTO, K. AND VAN DE GEIJN, R. A. 2002. On reducing TLB misses in matrix multiplication. Tech. rep. CS-TR-02-55, Department of Computer Sciences, The University of Texas at Austin.

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

GROPP, W. AND LUSK, E. 1994. User's guide for `mpich`, a portable implementation of MPI. Tech. rep. ANL-06/6, Argonne National Laboratory.

GROPP, W., LUSK, E., AND SKJELLUM, A. 1994. *Using MPI*. The MIT Press.

GULLBERG, G., HUESMAN, R., ROY, D. G., QI, J., AND REUTTER, B. 2003. Estimation of the parameter covariance matrix for a one-compartment cardiac perfusion model estimated from a dynamic sequence reconstructed using map iterative reconstruction algorithms. In *Nuclear Science Symposium Conference Record*. vol. 5. IEEE, 3019–3023.

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

GUNNELS, J. A., GUSTAVSON, F. G., HENRY, G. M., AND VAN DE GEIJN, R. A. 2001a. A family of high-performance matrix multiplication algorithms. In *Computational Science (ICCS'01), Part I*, V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. K. Tan, Eds. Lecture Notes in Computer Science, vol. 2073. Springer-Verlag, 51–60.

GUNNELS, J. A., KATZ, D. S., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2001b. Fault–tolerant high–performance matrix multiplication: Theory and practice. In *Proceedings of the International Conference for Dependable Systems and Networks (DSN'01)*. 47–56.

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

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

HIGHAM, N. J. 2002. *Accuracy and Stability of Numerical Algorithms*, 2nd Ed. Society for Industrial and Applied Mathematics, Philadelphia, PA.

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

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

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

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

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

QUINTANA, E. S., QUINTANA, G., SUN, X., AND VAN DE GEIJN, R. 2001. A note on parallel matrix inversion. *SIAM J. Sci. Comput. 22,* 5, 1762–1771.

SANSO, R. AND RUMMEL, R., Eds. 1989. Theory of satellite geodesy and gravity field determination. Lecture Notes in Earth Sciences, vol. 25. Springer-Verlag, Berlin, Germany.

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

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

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

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

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

Received July 2006; revised June 2007, October 2007; accepted October 2007

Updating an LU Factorization with Pivoting

ENRIQUE S. QUINTANA-ORTÍ
Universidad Jaime I
and
ROBERT A. VAN DE GEIJN
The University of Texas at Austin

We show how to compute an LU factorization of a matrix when the factors of a leading principle submatrix are already known. The approach incorporates pivoting akin to partial pivoting, a strategy we call *incremental pivoting*. An implementation using the Formal Linear Algebra Methods Environment (FLAME) application programming interface (API) is described. Experimental results demonstrate practical numerical stability and high performance on an Intel Itanium2 processor-based server.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra; G.4 [**Mathematics of Computing**]: Mathematical Software—*Efficiency*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: LU factorization, linear systems, updating, pivoting

ACM Reference Format:
Quintana-Ortí, E. S. and van de Geijn, R. A. 2008. Updating an LU factorization with pivoting. ACM Trans. Math. Softw. 35, 2, Article 11 (July 2008), 16 pages. DOI = 10.1145/1377612.1377615. http://doi.acm.org/10.1145/1377612.1377615.

This research was partially sponsored by NSF grants ACI-0305163, CCF-0342369 and CCF-0540926, and an equipment donation from Hewlett-Packard. Primary support for this work came from the J. Tinsley Oden Faculty Fellowship Research Program of the Institute for Computational Engineering and Sciences (ICES) at UT-Austin.
Authors' addresses: E. S. Quintana-Ortí, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaime I, 12.071 – Castellón, Spain; email: quintana@icc.uji.es. R. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: rvdg@cs.utexas.edu.

1. INTRODUCTION

In this article we consider the LU factorization of a nonsymmetric matrix A, partitioned as

$$A \rightarrow \left(\begin{array}{c|c} B & C \\ \hline D & E \end{array}\right) \tag{1}$$

when a factorization of B is to be reused as the other parts of the matrix change. This is known as the updating of an LU factorization.

Applications arising in boundary element methods (BEMs) often lead to very large, dense linear systems [Cwik et al. 1994; Geng et al. 1996]. For many of these applications the goal is to optimize a feature of an object. For example, BEMs may be used to model the radar signature of an airplane. In an effort to minimize this signature, it may be necessary to optimize the shape of a certain component of the airplane. If the degrees of freedom associated with this component are ordered last among all degrees of freedom, the matrix presents the structure given in Eq. (1). Now, as the shape of the component is modified, it is only the matrices C, D, and E that change together with the right-hand side vector of the corresponding linear system. Since the dimension of B is frequently much larger than those of the remaining three matrices, it is desirable to factorize B only once and to update the factorization as C, D, and E change. A standard LU factorization with partial pivoting does not provide a convenient solution to this problem, since the rows to be swapped during the application of the permutations may not lie only within B.

Little literature exists on this important topic. We have been made aware that an unblocked out-of-core (OOC) algorithm similar to our algorithm was reported in Yip [1979], but we have not been able to locate a copy of that report. The proposed addition of this functionality to LAPACK is discussed in Demmel and Dongarra [2005]. We already discussed preliminary results regarding the algorithm proposed in the current article in a conference paper [Joffrain et al. 2005], in which its application to OOC LU factorization with pivoting is the main focus.[1] In Gunter and van de Geijn [2005] the updating of a QR factorization via techniques, that are closely related to those proposed for the LU factorization in the current article, is reported.

The article is organized as follows: In Section 2 we review algorithms for computing the LU factorization with partial pivoting. In Section 3, we discuss how to update an LU factorization by considering the factorization of a 2×2 blocked matrix. The key insight of the work is found in this section: High-performance blocked algorithms can be synthesized by combining the pivoting strategies of LINPACK and LAPACK. Numerical stability is discussed in Section 4 and performance is reported in Section 5. Concluding remarks are given in the final section.

[1]More practical approaches to OOC LU factorization with partial pivoting exist [Toledo 1999; 1997; Toledo and Gustavson 1996; Klimkowski and van de Geijn 1995]. Therefore, OOC application of the approach is not further mentioned so as not to distract from the central message of this article.

We hereafter assume that the reader is familiar with Gauss transforms, their properties, and how they are used to factor a matrix. We start indexing elements of vectors and matrices at 0. Capital letters, lower-case letters, and lower-case Greek letters will be used to denote matrices, vectors, and scalars, respectively. The identity matrix of order n is denoted by I_n.

2. THE LU FACTORIZATION WITH PARTIAL PIVOTING

Given an $n \times n$ matrix A, its LU factorization with partial pivoting is given by $PA = LU$. Here P is a permutation matrix of order n, L is $n \times n$ unit lower triangular, and U is $n \times n$ upper triangular. We will denote the computation of P, L, and U by

$$[A, p] := [\{L\backslash U\}, p] = \mathrm{LU}(A), \tag{2}$$

where $\{L\backslash U\}$ is the matrix whose strictly lower triangular part equals L and whose upper triangular part equals U. Matrix L has ones on the diagonal, which need not be stored, and the factors L and U overwrite the original contents of A. The permutation matrix is generally stored in a vector p of n integers.

Solving the linear system $Ax = b$ now becomes a matter of solving $Ly = Pb$ followed by $Ux = y$. These two stages are referred to as *forward substitution* and *backward substitution*, respectively.

2.1 Unblocked Right-Looking LU Factorization

Two unblocked algorithms for computing the LU factorization with partial pivoting are given in Figure 1. There, $n(\cdot)$ stands for the number of columns of a matrix; the thick lines in the matrices/vectors denote how far computation has progressed; PIVOT(x) determines the element in x with largest magnitude, swaps this element with the top element, and returns the index of the element that was swapped; and $P(\pi_1)$ is the permutation matrix constructed by interchanging row 0 and row π_1 of the identity matrix. The dimension of a permutation matrix will not be specified since it is obvious from the context in which it is used. We believe the rest of the notation to be intuitive [Bientinesi and van de Geijn 2006; Bientinesi et al. 2005]. Both algorithms correspond to what is usually known as the right-looking variant. Upon completion, matrices L and U overwrite A. These algorithms also yield the LU factorization of a matrix with more rows than columns.

The LINPACK variant, $\mathrm{LU}^{\mathrm{LIN}}_{\mathrm{UNB}}$ hereafter, computes the LU factorization as a sequence of Gauss transforms interleaved with permutation matrices.

$$L_{n-1}\left(\begin{array}{c|c} I_{n-1} & 0 \\ \hline 0 & P(\pi_{n-1}) \end{array}\right)\cdots L_1\left(\begin{array}{c|c} 1 & 0 \\ \hline 0 & P(\pi_1) \end{array}\right) L_0 P(\pi_0) A = U$$

For the LAPACK variant $\mathrm{LU}^{\mathrm{LAP}}_{\mathrm{UNB}}$, it is recognized that by swapping those rows of matrix L that were already computed and stored to the left of the column that is currently being eliminated, the order of the Gauss transforms and

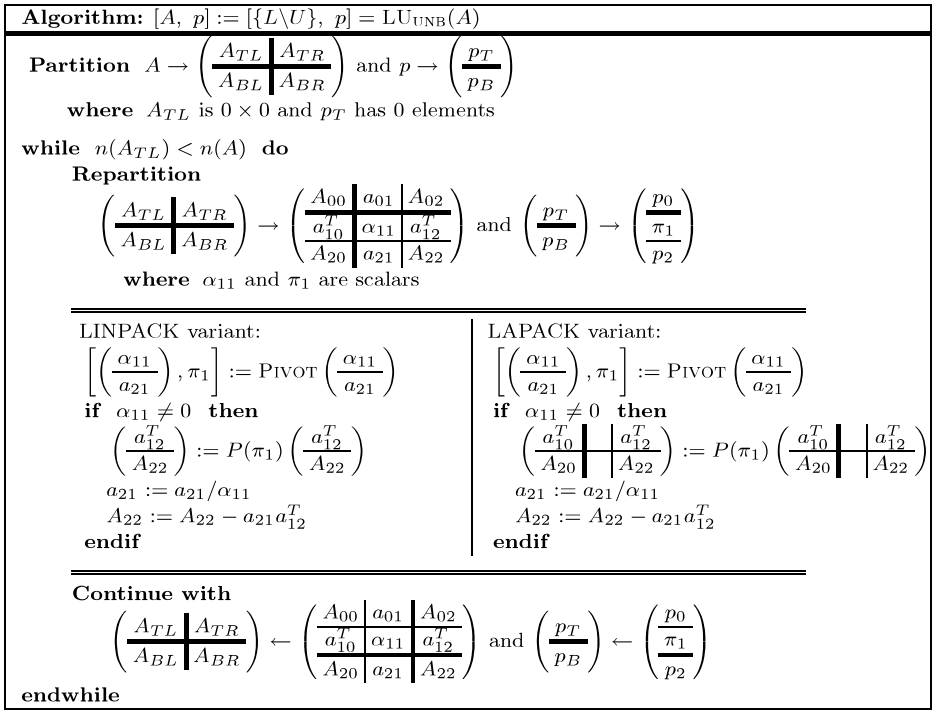


Fig. 1. LINPACK and LAPACK unblocked algorithms for the LU factorization.

permutation matrices can be rearranged so that $P(p)A = LU$. Here $P(p)$, with $p = \left(\begin{array}{c|c|c} \pi_0 & \cdots & \pi_{n-1} \end{array}\right)^T$, denotes the $n \times n$ permutation

$$\left(\begin{array}{c|c} I_{n-1} & 0 \\ \hline 0 & P(\pi_{n-1}) \end{array}\right) \cdots \left(\begin{array}{c|c} 1 & 0 \\ \hline 0 & P(\pi_1) \end{array}\right) P(\pi_0).$$

Both algorithms will execute to completion, even if an exact zero is encountered on the diagonal of U. This is important since it is possible that matrix B in (1) is singular even if A is not.

The difference between the two algorithms becomes most obvious when forward substitution is performed. For the LINPACK variant, forward substitution requires the application of permutations and Gauss transforms to be interleaved. For the LAPACK algorithm, the permutations are applied first on the right-hand side vector, after which a clean lower triangular solve yields the desired (intermediate) result: $Ly = P(p)b$. Depending on whether the LINPACK or LAPACK variant was used for LU factorization, we denote the forward-substitution stage, respectively, by $y := \mathrm{FS}^{\mathrm{LIN}}(A, p, b)$ or $y := \mathrm{FS}^{\mathrm{LAP}}(A, p, b)$, where A and p are assumed to contain the outputs of the corresponding factorization.

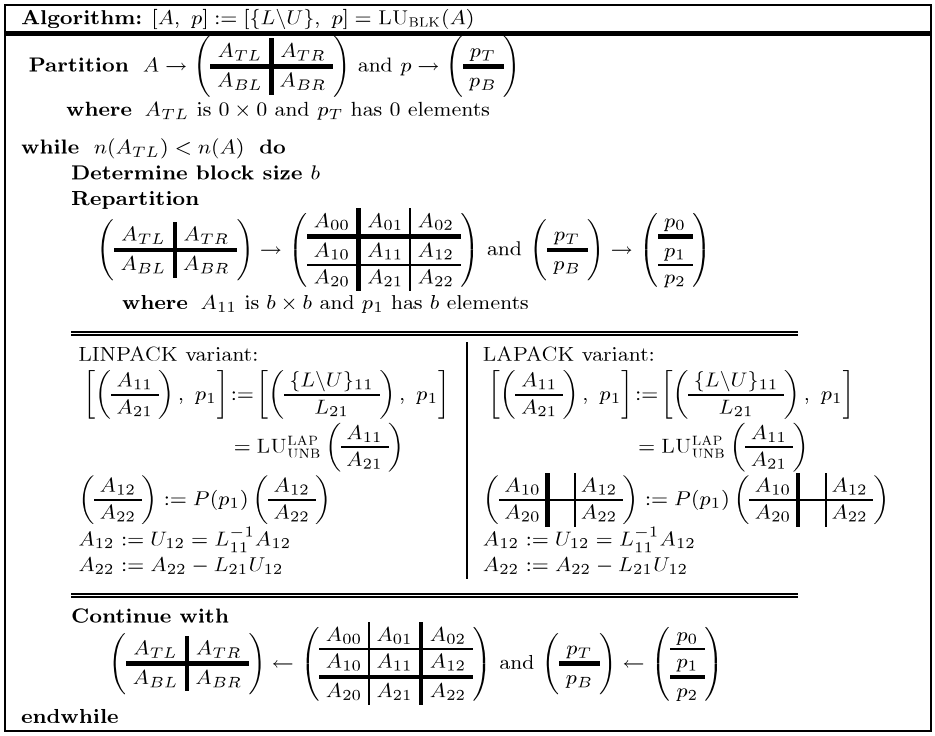


Fig. 2. LINPACK and LAPACK blocked algorithms for the LU factorization built upon an LAPACK unblocked factorization.

2.2 Blocked Right-Looking LU Factorization

It is well known that high performance can be achieved in a portable fashion by casting algorithms in terms of matrix-matrix multiplication [Kågström et al. 1998; 1995; Gustavson et al. 1998; Gunnels et al. 2001]. In Figure 2 we show LINPACK(-like) and LAPACK blocked algorithms $\mathrm{LU}^{\mathrm{LIN}}_{\mathrm{BLK}}$ and $\mathrm{LU}^{\mathrm{LAP}}_{\mathrm{BLK}}$, respectively, both built upon an LAPACK unblocked algorithm. The former algorithm really combines the LAPACK style of pivoting, within the factorization of a panel of width b, with the LINPACK style of pivoting. The two algorithms attain high performance on modern architectures with (multiple levels of) cache memory by casting the bulk of the computation in terms of the matrix-matrix multiplication $A_{22} := A_{22} - L_{21}U_{12}$, also called a rank-$k$ update, which is known to achieve high performance [Goto and van de Geijn 2008]. The algorithms also apply to matrices with more rows than columns.

As both LINPACK and LAPACK blocked algorithms are based on the LAPACK unblocked algorithm (which completes even if the current panel is singular), both will complete even for a singular matrix. If matrix A in Eq. (1) is nonsingular, then the upper triangular factor will also be nonsingular; this is what we need in order to use the factored matrix to solve a linear system.

3. UPDATING AN LU FACTORIZATION

In this section we discuss how to compute the LU factorization of the matrix in (1) in such a way that the LU factorization with partial pivoting of B can be reused if C, D, and E change. We consider A in Eq. (1) to be of dimension $n \times n$, with square B and E of orders n_B and n_E, respectively. For reference, factoring the matrix in (1) using standard LU factorization with partial pivoting costs $\frac{2}{3}n^3$ flops (floating-point arithmetic operations). In this expression (and future computational cost estimates) we neglect insignificant terms of lower-order complexity, including the cost of pivoting the rows.

3.1 Basic Procedure

We propose employing the following procedure, consisting of five steps, which computes an *LU factorization with incremental pivoting* of the matrix in Eq. (1).

Step 1: *Factor B.* Compute the LU factorization with partial pivoting

$$[B, p] := [\{L \backslash U\}, p] = \mathrm{LU}^{\mathrm{LAP}}_{\mathrm{BLK}}(B).$$

This step is skipped if B has already been factored. If the factors are to be used for future updates to C, D, and E, then a copy of U is needed since it is overwritten by subsequent steps.

Step 2: *Update C.* This is consistent with the factorization of B. This is

$$C := \mathrm{FS}^{\mathrm{LAP}}(B, p, C)$$

Step 3: *Factor* $\begin{pmatrix} U \\ D \end{pmatrix}$. Compute the LU factorization with partial pivoting

$$\left[\begin{pmatrix} U \\ D \end{pmatrix}, \bar{L}, r \right] := \left[\begin{pmatrix} \{\bar{L} \backslash \bar{U}\} \\ \check{L} \end{pmatrix}, r \right] = \mathrm{LU}^{\mathrm{LIN}}_{\mathrm{BLK}} \begin{pmatrix} U \\ D \end{pmatrix}.$$

Here, $\bar{U}$ overwrites the upper triangular part of B (where U was stored before this operation). The lower triangular matrix $\bar{L}$ that results needs to be stored separately, since both L, computed in step 1 and used at step 2, and $\bar{L}$ are needed during the forward-substitution stage when solving a linear system.

Step 4: *Update* $\begin{pmatrix} C \\ E \end{pmatrix}$. This is consistent with the factorization of $\begin{pmatrix} U \\ D \end{pmatrix}$.

$$\begin{pmatrix} C \\ E \end{pmatrix} := \mathrm{FS}^{\mathrm{LIN}} \left(\begin{pmatrix} \bar{L} \\ D \end{pmatrix}, r, \begin{pmatrix} C \\ E \end{pmatrix} \right)$$

Step 5: *Factor E.* Finally, compute the LU factorization with partial pivoting

$$[E, s] := \left[\{\check{L} \backslash \check{U}\}, s \right] = \mathrm{LU}^{\mathrm{LAP}}_{\mathrm{BLK}}(E).$$

Overall, the five steps of the procedure apply Gauss transforms and permutations to reduce A to an upper triangular matrix, as

$$\left(\begin{array}{c|c} I & 0 \\ \hline 0 & \tilde{L}^{-1}P(s) \end{array}\right) \left(\begin{array}{c|c} \bar{L} & 0 \\ \hline \check{L} & I \end{array}\right)^{-1} P(r) \underbrace{\left(\begin{array}{c|c} L^{-1}P(p) & 0 \\ \hline 0 & I \end{array}\right) \left(\begin{array}{c|c} B & C \\ \hline D & E \end{array}\right)}_{\text{steps 1 and 2}} =$$

$$\left(\begin{array}{c|c} I & 0 \\ \hline 0 & \tilde{L}^{-1}P(s) \end{array}\right) \underbrace{\left(\begin{array}{c|c} \bar{L} & 0 \\ \hline \check{L} & I \end{array}\right)^{-1} P(r) \left(\begin{array}{c|c} U & \hat{C} \\ \hline D & E \end{array}\right)}_{\text{steps 3 and 4}} =$$

$$\underbrace{\left(\begin{array}{c|c} I & 0 \\ \hline 0 & \tilde{L}^{-1}P(s) \end{array}\right) \left(\begin{array}{c|c} \bar{U} & \check{C} \\ \hline 0 & \check{E} \end{array}\right)}_{\text{step 5}} = \left(\begin{array}{c|c} \bar{U} & \check{C} \\ \hline 0 & \tilde{U} \end{array}\right),$$

where $\{L\backslash U\}$, $\left\{\left(\begin{array}{c|c} \bar{L} & 0 \\ \hline \check{L} & I \end{array}\right) \backslash \left(\begin{array}{c} \bar{U} \\ \hline 0 \end{array}\right)\right\}$, and $\{\tilde{L}\backslash\tilde{U}\}$ are the triangular factors computed, respectively, in the LU factorizations in steps 1, 3, and 5; p, r, and s are the corresponding permutation vectors; $\hat{C}$ is the matrix that results from overwriting C with $L^{-1}P(p)C$; and $\left(\begin{array}{c} \check{C} \\ \hline \check{E} \end{array}\right)$ are the blocks that result from $\left(\begin{array}{c|c} I & 0 \\ \hline 0 & \tilde{L}^{-1}P(s) \end{array}\right) \left(\begin{array}{c} \hat{C} \\ \hline E \end{array}\right)$.

3.2 Analysis of the Basic Procedure

For now, the factorization in step 3 does not take advantage of any zeroes below the diagonal of U: After matrix B is factored and C is updated, the matrix $\left(\begin{array}{c|c} U & C \\ \hline D & E \end{array}\right)$ is factored as if it is a matrix without special structure. Its cost is stated in the column labeled "Basic procedure" in Table I. There we only report significant terms: We assume that $b \ll n_E, n_B$ and report only those costs that equal at least $O(bn_En_B)$, $O(bn_E^2)$, or $O(bn_B^2)$. *If* n_E is *small* (i.e., $n_B \approx n$), the procedure clearly does not benefit from the existence of an already factored B. Also, the procedure requires additional storage for the $n_B \times n_B$ lower triangular matrix $\bar{L}$ computed in step 3.

Table I. Computational Cost (in flops) of Different Approaches to Compute LU Factorization of the Matrix in Eq. (1).

Operation	Approximate cost (in flops)		
	Basic procedure	Structure-Aware LAPACK procedure	Structure-Aware LINPACK procedure
1: Factor B	$\frac{2}{3}n_B^3$	$\frac{2}{3}n_B^3$	$\frac{2}{3}n_B^3$
2: Update C	$n_B^2 n_E$	$n_B^2 n_E$	$n_B^2 n_E$
3: Factor $\begin{pmatrix} U \\ D \end{pmatrix}$	$n_B^2 n_E$ $+\frac{2}{3}n_B^3$	$n_B^2 n_E$ $+\frac{1}{2}b n_B^2$	$n_B^2 n_E$ $+\frac{1}{2}b n_B^2$
4: Update $\begin{pmatrix} C \\ E \end{pmatrix}$	$2n_B n_E^2$ $+n_B^2 n_E$	$2n_B n_E^2$ $+n_B^2 n_E$	$2n_B n_E^2$ $+b n_B n_E$
5: Factor E	$\frac{2}{3}n_E^3$	$\frac{2}{3}n_E^3$	$\frac{2}{3}n_E^3$
Total	$\frac{2}{3}n^3$ $+\frac{2}{3}n_B^3 + n_B^2 n_E$	$\frac{2}{3}n^3$ $+n_B^2\left(\frac{1}{2}b + n_E\right)$	$\frac{2}{3}n^3$ $+b n_B\left(\frac{n_B}{2} + n_E\right)$

The highlighted costs are those incurred in excess of the cost of a standard LU factorization.

We describe next how to reduce both the computational and storage requirements by exploiting the upper triangular structure of U during steps 3 and 4.

3.3 Exploiting the Structure in Step 3

A blocked algorithm that exploits the upper triangular structure of U is given in Figure 3 and illustrated in Figure 4. We name this algorithm $\text{LU}_{\text{BLK}}^{\text{SA-LIN}}$ to reflect that it computes a "structure-aware" (SA) LU factorization. At each iteration of the algorithm, the panel of b columns consisting of $\begin{pmatrix} U_{11} \\ D_1 \end{pmatrix}$ is factored using the LAPACK unblocked algorithm $\text{LU}_{\text{UNB}}^{\text{LAP}}$. (In our implementation this algorithm is modified to also take advantage of the zeroes below the diagonal of U_{11}.) As part of the factorization, U_{11} is overwritten by $\{\bar{L}_1 \backslash \bar{U}_{11}\}$. However, in order to preserve the strictly lower triangular part of U_{11} (where part of the matrix L, that was computed in step 1, is stored), we employ the $b \times b$ submatrix $\bar{L}_1$ of the $n_B \times b$ array $\bar{L}$ (see Figure 3). As in the LINPACK blocked algorithm in Figure 2, the LAPACK and LINPACK styles of pivoting are combined: The columns of the current panel are pivoted using the LAPACK approach, but the permutations from this factorization are only applied to $\begin{pmatrix} U_{12} \\ D_2 \end{pmatrix}$.

The cost of this approach is given in step 3 of the column labeled "Structure-Aware LINPACK procedure" in Table I. The cost difference comes from the updates of U_{12} shown in Figure 3, and provided $b \ll n_B$, is insignificant compared to $\frac{2}{3}n^3$.

An SA LAPACK blocked algorithm for step 3 only differs from that in Figure 3 in that, at a certain iteration after the LU factorization of the current panel is computed, these permutations have to be applied to $\begin{pmatrix} U_{10} \\ D_0 \end{pmatrix}$ as well. As indicated in step 3 of the column labeled "Structure-Aware LAPACK procedure," this does not incur extra cost for *this* step. However, it does require an $n_B \times n_B$ array for storing $\bar{L}$ (see Figure 4) and, as we will see next, makes step 4 more expensive. On the other hand, the SA LINPACK algorithm only

Algorithm: $\left[\left(\frac{U}{D}\right), \bar{L}, r\right] := \mathrm{LU}^{\mathrm{SA-LIN}}_{\mathrm{BLK}}\left(\frac{U}{D}\right)$

Partition $U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)$, $D \rightarrow \left(\begin{array}{c|c} D_L & D_R \end{array}\right)$, $\bar{L} \rightarrow \left(\frac{\bar{L}_T}{\bar{L}_B}\right)$, $r \rightarrow \left(\frac{r_T}{r_B}\right)$

where U_{TL} is 0×0, D_L has 0 columns, $\bar{L}_T$ has 0 rows, and r_T has 0 elements

while $n(U_{TL}) < n(U)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right),$$

$$\left(\frac{\bar{L}_T}{\bar{L}_B}\right) \rightarrow \left(\frac{\bar{L}_0}{\bar{L}_1}{\bar{L}_2}\right), \left(\frac{r_T}{r_B}\right) \rightarrow \left(\frac{r_0}{r_1}{r_2}\right)$$

where U_{11} is $b \times b$, D_1 has b columns, $\bar{L}_1$ has b rows, and r_1 has b elements

---

$\left[\left(\frac{\{\bar{L}_1 \backslash U_{11}\}}{D_1}\right), r_1\right] := \mathrm{LU}^{\mathrm{LAP}}_{\mathrm{UNB}}\left(\frac{U_{11}}{D_1}\right)$

$\left(\frac{U_{12}}{D_2}\right) := P(r_1)\left(\frac{U_{12}}{D_2}\right)$

$U_{12} := \bar{L}_1^{-1} U_{12}$

$D_2 := D_2 - D_1 U_{12}$

Continue with

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right),$$

$$\left(\frac{\bar{L}_T}{\bar{L}_B}\right) \leftarrow \left(\frac{\bar{L}_0}{\bar{L}_1}{\bar{L}_2}\right), \left(\frac{r_T}{r_B}\right) \leftarrow \left(\frac{r_0}{r_1}{r_2}\right)$$

endwhile

Fig. 3. SA-LINPACK blocked algorithm for the LU factorization of $\left(U^T,\ D^T\right)^T$ built upon an LAPACK blocked factorization.

requires an $n_B \times b$ additional work space for storing the factors, as indicated in Figure 4.

3.4 Revisiting the Update in Step 4

The same optimizations made in step 3 must now be carried over to the update of $\left(\frac{C}{E}\right)$. The algorithm for this is given in Figure 5. Computation corresponding to zeroes is avoided so that the cost of performing the update is $2n_B n_E^2 + b\, n_B n_E$ flops, as indicated in step 4 of Table I.

Applying the SA LAPACK blocked algorithm in step 3 destroys the structure of the lower triangular matrix, which cannot be recovered during the forward substitution stage in step 4. This explains the additional cost reported for this variant in Table I.

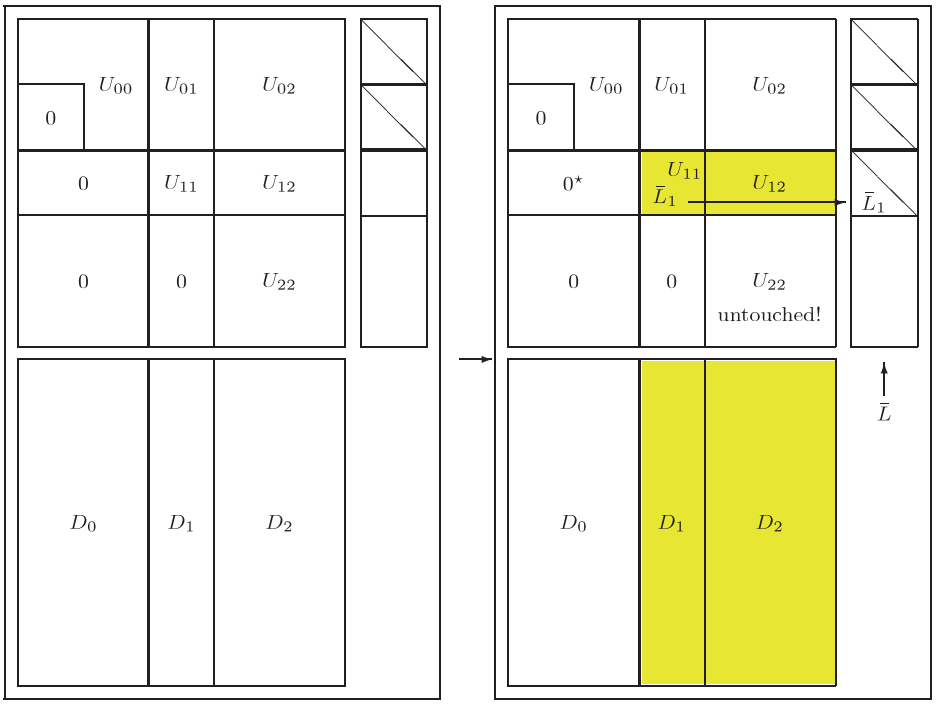


Fig. 4. Illustration of an iteration of the SA LINPACK blocked algorithm used in step 3 and how it preserves most of the zeroes in U. The zeroes below the diagonal are preserved, except within the $b \times b$ diagonal blocks, where pivoting will fill below the diagonal. The shaded areas are the ones updated as part of the current iteration. The fact that U_{22} is not updated demonstrates how computation can be reduced. If the SA LAPACK blocked algorithm was used, then nonzeroes would appear during this iteration in the block marked as $0^\star$, due to pivoting; as a result, upon completion, zeros would be lost in the full strictly lower triangular part of U.

3.5 Key Contribution

The difference in cost of the three different approaches analyzed in Table I is illustrated in Figure 6. It reports the ratios in cost of the aforesaid different procedures and that of the LU factorization with partial pivoting for a matrix with $n_B = 1000$ and different values of n_E, using $b = 32$. The analysis shows that the overhead of the SA LINPACK procedure is consistently low. On the other hand, as $n_E/n \to 1$ the cost of the basic procedure, which is initially twice as expensive as that of the LU factorization with partial pivoting, is decreased. The SA LAPACK procedure only presents a negligible overhead when $n_E \to 0$, that is, when the dimension of the update is very small.

The key insight of the proposed approach is the recognition that combining LINPACK- and LAPACK-style pivoting allows one to use a blocked algorithm while avoiding filling most of the zeroes in the lower triangular part of U. This, in turn, makes the extra cost of step 4 acceptable. In other words, for the SA LINPACK procedure, the benefit of higher performance of the blocked algorithm comes at the expense of a lower-order amount of extra computation.

Algorithm: $\left[\left(\frac{C}{E}\right)\right] := \mathrm{FS}_{\mathrm{BLK}}^{\mathrm{SA-LIN}}\left(\left(\frac{\bar{L}}{D}\right), r, \left(\frac{C}{E}\right)\right)$

Partition $\bar{L} \rightarrow \left(\frac{\bar{L}_T}{\bar{L}_B}\right)$, $D \rightarrow \left(\begin{array}{c|c} D_L & D_R \end{array}\right)$, $r \rightarrow \left(\frac{r_T}{r_B}\right)$, $C \rightarrow \left(\frac{C_T}{C_B}\right)$,

where $\bar{L}_T$ and C_T have 0 rows, D_L has 0 columns, and r_T has 0 elements

while $n(D_L) < n(D)$ **do**

Determine block size b

Repartition

$$\left(\frac{\bar{L}_T}{\bar{L}_B}\right) \rightarrow \left(\frac{\bar{L}_0}{\bar{L}_1}{\bar{L}_2}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right),$$

$$\left(\frac{r_T}{r_B}\right) \rightarrow \left(\frac{r_0}{r_1}{r_2}\right), \left(\frac{C_T}{C_B}\right) \rightarrow \left(\frac{C_0}{C_1}{C_2}\right),$$

where $\bar{L}_1$ and C_1 have b rows, D_1 has b columns, and r_1 has b elements

---

$\left(\frac{C_1}{E}\right) := P(r_1)\left(\frac{C_1}{E}\right)$

$C_1 := \bar{L}_1^{-1} C_1$

$E := E - D_1 C_1$

Continue with

$$\left(\frac{\bar{L}_T}{\bar{L}_B}\right) \leftarrow \left(\frac{\bar{L}_0}{\bar{L}_1}{\bar{L}_2}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right),$$

$$\left(\frac{r_T}{r_B}\right) \leftarrow \left(\frac{r_0}{r_1}{r_2}\right), \left(\frac{C_T}{C_B}\right) \leftarrow \left(\frac{C_0}{C_1}{C_2}\right),$$

endwhile

Fig. 5. SA-LINPACK blocked algorithm for the update of $\left(C^T,\ E^T\right)^T$, consistent with the SA-LINPACK blocked LU factorization of $\left(U^T,\ D^T\right)^T$.

The extra memory for the SA LINPACK procedure consists of an $n_B \times n_B$ upper triangular matrix and an $n_B \times b$ array.

4. REMARKS ON NUMERICAL STABILITY

The algorithm for the LU factorization with incremental pivoting carries out a sequence of row permutations (corresponding to the application of permutations) which are different from those that would be performed in an LU factorization with partial pivoting. Therefore, the numerical stability of this algorithm is also different. In this section we provide some remarks on the stability of the new algorithm. We note that all three procedures described in the previous section (basic, SA LINPACK, and SA LAPACK) perform the same sequence of row permutations.

The numerical (backward) stability of an algorithm that computes the LU factorization of a matrix A depends on the growth factor [Stewart 1998]

$$\rho = \frac{\|L\|\|U\|}{\|A\|}, \tag{3}$$

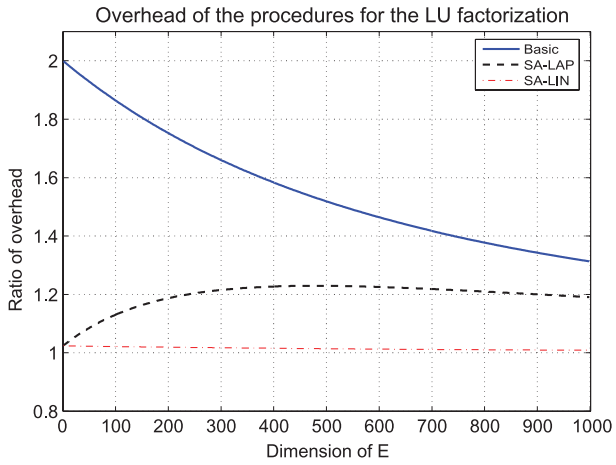


Fig. 6. Overhead cost of the different approaches to compute the LU factorization in Eq. (1) with respect to the cost of the LU factorization with partial pivoting.

which is basically determined by problem size and pivoting strategy. For example, the growth factors of complete, partial, and *pairwise* [Wilkinson 1965, p. 236] pivoting have been demonstrated bounded as $\rho_c \leq n^{1/2}(2 \cdot 3^{1/2} \cdots n^{1/n-1})$, $\rho_p \leq 2^{n-1}$, and $\rho_w \leq 4^{n-1}$, respectively [Sorensen 1985; Stewart 1998]. Statistical models and extensive experimentations in Trefethen and Schreiber [1990] showed that, on average, $\rho_c \approx n^{1/2}$, $\rho_p \approx n^{2/3}$, and $\rho_w \approx n$, inferring that in practice both partial and pairwise pivoting are numerically stable, and pairwise pivoting can be expected to numerically behave only slightly worse than partial pivoting.

The new algorithm applies partial pivoting during the factorization of B and then again in the factorization of $\begin{pmatrix} U \\ D \end{pmatrix}$. This can be considered as a blocked variant of pairwise pivoting. Thus, we can expect an element growth for the algorithm that is between those of partial and pairwise pivoting. Next we elaborate an experiment that provides evidence in support of this observation.

In Figure 7 we report the element growths observed during computation of the LU factorization of matrices as in Eq. (1), with $n_B = 100$ and dimensions for E ranging from $n_E = 5$ to 100 using partial, incremental, and pairwise pivoting. The entries of the matrices are generated randomly, chosen from a uniform distribution in the interval (0.0, 1.0). The experiment was carried out on an Intel Xeon processor using Matlab® 7.0.4 (IEEE double-precision arithmetic). The results report the average element growth for 100 different matrices for each matrix dimension. The figure shows the growth factor of incremental pivoting to be smaller than that of pairwise pivoting and to approximate that of partial pivoting. A similar behavior was obtained for other matrix types: uniform distribution in $(-1.0, 1.0)$, normal distribution with mean 0.0 and deviation 1.0 (N[0.0, 1.0]), symmetric matrices with elements in N[0.0, 1.0], and Toeplitz matrices with elements in N[0.0, 1.0]. Only for orthogonal matrices with Haar distribution [Trefethen and Schreiber 1990] did we obtain significantly

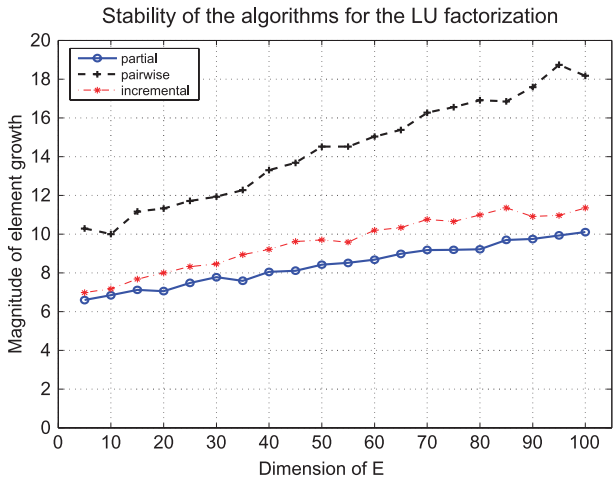


Fig. 7. Element growth in LU factorization using different pivoting techniques.

different results. In that case, incremental pivoting attained a smaller element growth than pairwise pivoting, and both outperformed the element growth of partial pivoting. Explaining the behavior of this case is beyond the scope of this work.

For those who are not sufficiently satisfied with the element growth of incremental pivoting, we propose to perform a few refinement iterations of the solution to $Ax = b$ at a cost of $O(n^2)$ flops per step, as this guarantees stability at a low computational cost [Higham 2002].

5. PERFORMANCE

In this section we report results for a high-performance implementation of the SA LINPACK procedure.

5.1 Implementation

The FLAME library (version 0.9) was used to implement a high-performance LU factorization with partial pivoting and the SA LINPACK procedure. The benefit of this API is that the code closely resembles the algorithms as they are presented in Figures 1 through 3 and 5. The performance of the FLAME LU factorization with partial pivoting is highly competitive with LAPACK and vendor implementations of this operation.

The implementations can be examined by visiting http://www.cs.utexas.edu/users/flame/Publications/. For further information on FLAME, visit www.cs.utexas.edu/users/flame.

5.2 Platform

Performance experiments were performed in double-precision arithmetic on an Intel Itanium2 (1.5 GHz) processor-based workstation capable of attaining 6 GFLOPS (10^9 flops per second). For reference, the algorithm for the FLAME

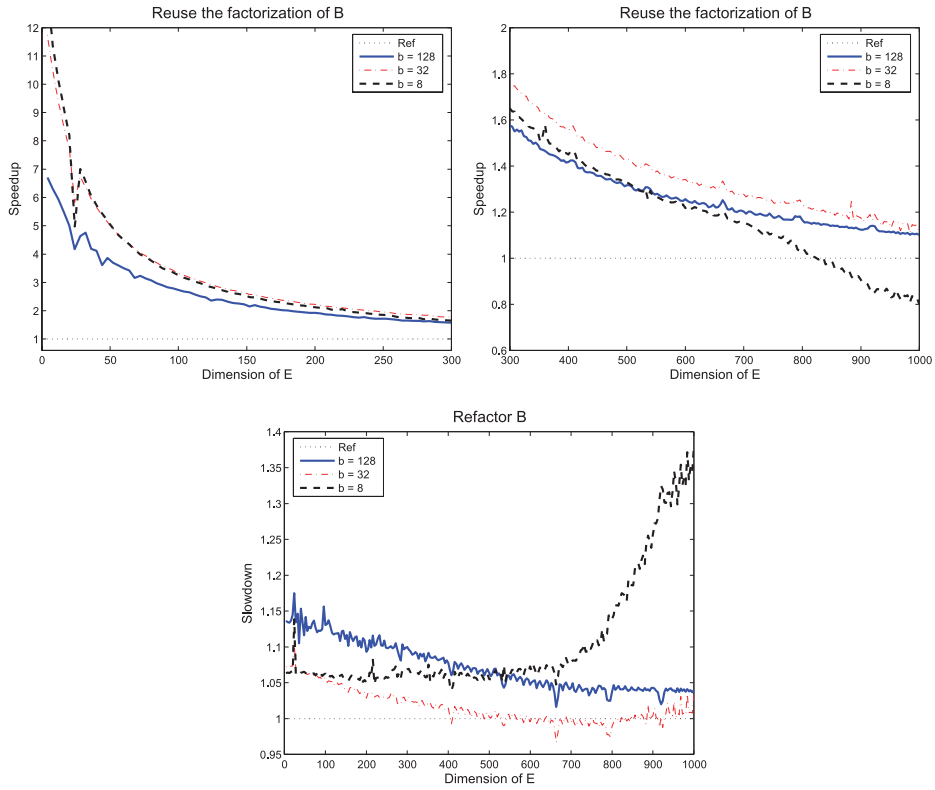


Fig. 8. Top: speedup attained when B is not refactored, over LU factorization with partial pivoting of the entire matrix; bottom: slowdown for the first factorization (when B must also be factored).

LU factorization with partial pivoting delivered 4.8 GFLOPS for a 2000×2000 matrix. A block size $b = 128$ was employed in this procedure for all experiments reported next. The implementation was linked to the GotoBLAS R1.6 basic linear algebra subprograms (BLAS) library [Goto 2004]. The BLAS routine DGEMM which is used to compute $C := C - AB$ ($C \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{m \times k}$, and $B \in \mathbb{R}^{k \times n}$) attains the best performance when the common dimension of A and B, namely k, is equal to 128. Notice that most computation in the SA LINPACK procedure is cast in terms of this operation, with $k = b$.

The performance benefits reported on this platform are representative of those that can be expected on other current architectures.

5.3 Results

In Figure 8 (top) we show the speedup attained when an existing factorization of B is reused, by reporting the time required to factor Eq. (1) with high-performance LU factorization with partial pivoting divided by the time required to update an existing factorization of B via the SA LINPACK procedure (steps 2 through 5). In that figure, $n_B = 1000$ and n_E is varied from 0 to 1000. The results are reported when different block size b's are chosen. The

DGEMM operation, in terms of which most computation is cast, attains the best performance when $b = 128$ is chosen. However, this generates enough additional flops that the speedup is higher when b is chosen smaller. When n_E is very small, $b = 8$ (for steps 2 through 5) yields the best performance. As n_E increases, performance improves by choosing $b = 32$ (for steps 2 through 5).

The effect of the overhead of the extra computations is demonstrated in Figure 8 (bottom). There, we report the ratio of the time required by steps 1 through 5 of the SA LINPACK procedure divided by the time required by LU factorization with partial pivoting of Eq. (1). The results in the figure may be somewhat disturbing: The algorithm that views the matrix as four quadrants attains as good or even better performance than the algorithm that views the matrix as a single unit and performs less computation. The likely explanation is that the standard LU factorization would also benefit from a variable block size as the problem size changes, rather than fixing it at $b = 128$. We did not further investigate this issue, since we did not want to make raw performance the primary focus of the article.

6. CONCLUSIONS

We have proposed blocked algorithms for updating an LU factorization. They have been shown to attain high performance and to greatly reduce the cost of an update to a matrix for which a partial factorization already exists. The key insight is the synthesis of LINPACK- and LAPACK-style pivoting. While some additional computation is required, this is more than offset by the improvement in performance that comes from casting computation in terms of matrix-matrix multiplication.

We acknowledge that the question of the numerical stability of the new algorithm relative to that of LU factorization with partial pivoting remains open. Strictly speaking, LU factorization with partial pivoting is itself not numerically stable, but practical experience has shown be effective in practice. Theoretical results that rigorously bound the additional element growth are in order, but are beyond the scope of the present article.

REFERENCES

BIENTINESI, P., GUNNELS, J. A., MYERS, M. E., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2005. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Softw. 31,* 1 (Mar.), 1–26.

BIENTINESI, P. AND VAN DE GEIJN, R. 2006. Representing dense linear algebra algorithms: A farewell to indices. Tech. Rep. FLAME Working Note 17, CS-TR-2006-10, Department of Computer Sciences, The University of Texas at Austin.

CWIK, T., VAN DE GEIJN, R., AND PATTERSON, J. 1994. The application of parallel computation to integral equation models of electromagnetic scattering. *J. Optic. Soc. Amer. A 11,* 4 (Apr.), 1538–1545.

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

GENG, P., ODEN, J. T., AND VAN DE GEIJN, R. 1996. Massively parallel computation for acoustical scattering problems using boundary element methods. *J. Sound Vibra. 191,* 1, 145–165.

GOTO, K. 2004. TACC software and tools. http://www.tacc.utexas.edu/resources/software/.

GOTO, K. AND VAN DE GEIJN, R. A. 2008. Anatomy of a high-performance matrix multiplication. *ACM Trans. Math. Softw. 34,* 3 (to appear).

GUNNELS, J. A., HENRY, G. M., AND VAN DE GEIJN, R. A. 2001. A family of high-performance matrix multiplication algorithms. In *Proceedings of the International Conference on Computational Science (ICCS), Part I*, V. N. Alexandrov et al., eds. Lecture Notes in Computer Science, vol. 2073. Springer, 51–60.

GUNTER, B. AND VAN DE GEIJN, R. 2005. Parallel out-of-core computation and updating of the QR factorization. *ACM Trans. Math. Softw. 31,* 1 (Mar.), 60–78.

GUSTAVSON, F., HENRIKSSON, A., JONSSON, I., KÅGSTRÖM, B., AND LING, P. 1998. Superscalar GEMM-based level 3 BLAS – The on-going evolution of a portable and high-performance library. In *Proceedings of the Workshop Applied Parallel Computing (PARA), Large Scale Scientific and Industrial Problems*, B. K. et al., eds. Lecture Notes in Computer Science, vol. 1541. Springer, 207–215.

HIGHAM, N. J. 2002. *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Society for Industrial and Applied Mathematics, Philadelphia, PA.

JOFFRAIN, T., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2005. Rapid development of high-performance out-of-core solvers. In *Proceedings of the Workshop on Applied Parallel Computing (PARA 2004)*, J. Dongarra et al., eds. Lecture Notes in Computer Science, vol. 3732. Springer, 413–422.

KÅGSTRÖM, B., LING, P., AND LOAN, C. V. 1995. Gemm-Based level 3 blas: High-Performance model, implementations and performance evaluation benchmark. LAPACK Working Note no. 107 CS-95-315, University of Tennessee. November.

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

KLIMKOWSKI, K. AND VAN DE GEIJN, R. 1995. Anatomy of an out-of-core dense linear solver. In *Proceedings of the International Conference on Parallel Processing*. vol. III - *Algorithms and Applications*, 29–33.

SORENSEN, D. C. 1985. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Trans. Comput. C-34,* 3, 274–278.

STEWART, G. W. 1998. *Matrix Algorithms. Volume I: Basic Decompositions*. SIAM, Philadelphia, PA.

TOLEDO, S. 1997. Locality of reference in LU decomposition with partial pivoting. *SIAM J. Matrix Anal. Appl. 18,* 4, 1065–1081.

TOLEDO, S. 1999. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms and Visualization*, J. Abello and J. S. Vitter, eds. American Mathematical Society Press, Providence, RI, 161–180.

TOLEDO, S. AND GUSTAVSON, F. 1996. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Proceedings of the 4th Workshop on I/O in Parallel and Distributed Systems*, 28–40.

TREFETHEN, L. N. AND SCHREIBER, R. S. 1990. Average-Case stability of Gaussian elimination. *SIAM J. Matrix Anal. Appl. 11,* 3, 335–360.

WILKINSON, J. H. 1965. *The Algebraic Eigenvalue Problem*. Oxford University Press, London.

YIP, E. L. 1979. Fortran subroutines for out-of-core solutions of large complex linear systems. Tech. Rep. CR-159142, NASA.

Received August 2006; revised December 2007; accepted December 2007

Scalable Parallelization of FLAME Code via the Workqueuing Model

FIELD G. VAN ZEE
The University of Texas at Austin
PAOLO BIENTINESI
Duke University
and
TZE MENG LOW and ROBERT A. VAN DE GEIJN
The University of Texas at Austin

We discuss the OpenMP parallelization of linear algebra algorithms that are coded using the Formal Linear Algebra Methods Environment (FLAME) API. This API expresses algorithms at a higher level of abstraction, avoids the use loop and array indices, and represents these algorithms as they are formally derived and presented. We report on two implementations of the workqueuing model, neither of which requires the use of explicit indices to specify parallelism. The first implementation uses the experimental `taskq` pragma, which may influence the adoption of a similar construct into OpenMP 3.0. The second workqueuing implementation is domain-specific to FLAME but allows us to illustrate the benefits of sorting tasks according to their computational cost prior to parallel execution. In addition, we discuss how scalable parallelization of dense linear algebra algorithms via OpenMP will require a two-dimensional partitioning of operands much like a 2D data distribution is needed on distributed memory architectures. We illustrate the issues and solutions by discussing the parallelization of the symmetric rank-k update and report impressive performance on an SGI system with 14 Itanium2 processors.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: FLAME, OpenMP, SMP, parallel, scalability, workqueuing

This research was partially sponsored by NSF grants CCF-0540926, CCF-0342369, and ACI 0305163. In addition, Dr. James Truchard (National Instruments) graciously made an unrestricted donation to our research.

Authors' addresses: F. G. Van Zee, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: `field@cs.utexas.edu`; P. Bientinesi, Department of Computer Science, Duke University, Durham, NC 27708; email: `pauldj@cs.duke.edu`; T. M. Low, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: `ltm@cs.utexas.edu`; R. A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: `rvdg@cs.utexas.edu`.

ACM Reference Format:

Van Zee, F. G., Bientinesi, P., Low, T. M., van de Geijn, R. A. 2008. Scalable parallelization of FLAME code via the workqueuing model. ACM Trans. Math. Softw. 34, 2, Article 10 (March 2008), 29 pages. DOI = 10.1145/1326548.1326552 http://doi.acm.org/ 10.1145/1326548.1326552

1. INTRODUCTION

FLAME. The Formal Linear Algebra Methods Environment (FLAME) project pursues a systematic methodology for deriving and implementing linear algebra libraries.The methodology is goal-oriented: given a mathematical specification of the operation to be implemented, prescribed steps yield a family of algorithms for computing the operation. A proof of correctness is also given as part of the derivation. The resulting algorithms are expressed at a high level of abstraction, much like one would present algorithms with pseudocode in a classroom setting [Bientinesi et al. 2005; Bientinesi 2006]. Application Programming Interfaces (APIs) allow the code to closely resemble the formal algorithm structure, thereby reducing the opportunity for the introduction of bugs in the translation from algorithm to implementation. APIs have been defined for the Matlab M-script language, for the C and Fortran programming languages, and even as an extension to the Parallel Linear Algebra Package (PLAPACK) [van de Geijn 1997; Bientinesi et al. 2005]. The scope of FLAME includes the Basic Linear Algebra Subprograms (BLAS) [Lawson et al. 1979; Dongarra et al. 1988, 1990], many operations from LAPACK [Anderson et al. 1992], and a large number of operations encountered in Control Theory [Quintana-Ortí and van de Geijn 2003; Bientinesi 2006].

The workqueuing model. Shah et al. [1999] proposed the workqueuing model specifically to overcome the limitations of the OpenMP `for` and `sections` constructs. In OpenMP, workqueuing parallelism would be derived through the use of two new directives: `taskq` and `task`. The workqueuing model offers distinct advantages over conventional parallel OpenMP constructs. Namely, workqueuing provides a method of parallelizing loops that abstracts completely from array and loop indexing; instead, the model is work-oriented, allowing the programmer to parallelize independent units of computation created within `for` and `while` statements. The virtues of workqueuing and the clean abstraction of FLAME allow the programmer to quickly parallelize any FLAME algorithm whose subproblems exhibit no interdependencies.

High performance. Contrary to conventional wisdom, elegant algorithms need not compromise on performance. Figure 1 gives the reader a general idea of the performance that we can attain in our algorithms with minimal effort. It is worth noting that while FLAME well outperforms the Intel Math Kernel Library (MKL), it is edged out by GotoBLAS for smaller problem sizes. However, this is not surprising. Kazushige Goto, author of GotoBLAS [Goto 2006], frequently collaborates with the FLAME project. Our open exchange of ideas allows him to combine our best findings with his own at a lower, more architecture-aware level.

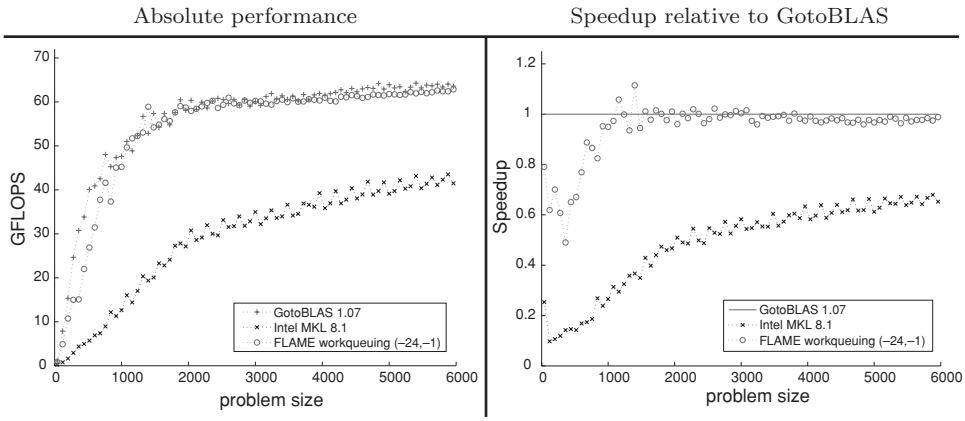


Fig. 1. Performance of parallel SYRK implementations (12 threads on 12 Itanium2 CPUs) using GotoBLAS 1.07, Intel MKL 8.1, and FLAME Variant 2 (parallelized with FLAME workqueuing) when m equals the problem size and $k = 200$. Further details of these experiments may be found in Section 6. *Bottom line:* An elegantly coded FLAME algorithm delivers impressive performance gains over a major vendor math library and quickly converges to perform on par with the cutting-edge GotoBLAS implementation.

Contributions. This article makes the following contributions:

—We show how algorithms written with the FLAME/C API can be naturally parallelized for Symmetric Multi-Processor (SMP) systems by employing the workqueuing model.

—We demonstrate the general applicability of the approach with a concrete example: the computation of the symmetric rank-k update (SYRK) operation. This operation is supported by the BLAS and is important in higher-level operations such as the Cholesky factorization.

—Implementations are given as a case study for the experimental OpenMP `taskq` pragma and for workqueuing as a method of obtaining parallelism in general.

—A custom workqueuing mechanism is described that allows algorithms encoded with the FLAME/C API to be parallelized via the workqueuing model without relying on the `taskq` pragma, which is currently only implemented within the Intel compilers.

—A compelling argument is made for the need to provide the programmer with more control over task scheduling within the OpenMP workqueuing interface.

—Load-balancing issues are discussed that provide insight into the interplay between different algorithmic variants for computing the same linear algebra operations and the order in which tasks are enqueued.

—A case is made that a 2D work distribution is required for scalability on SMP systems with large numbers of processors, much like a 2D data and work distribution is required on distributed memory architectures.

$C = \left(\begin{array}{c\|c} A_T A_T^T + \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array}\right)$ Loop-invariant 1	$C = \left(\begin{array}{c\|c} A_T A_T^T + \hat{C}_{TL} & \star \\ \hline A_B A_T^T + \hat{C}_{BL} & \hat{C}_{BR} \end{array}\right)$ Loop-invariant 2
$C = \left(\begin{array}{c\|c} \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & A_B A_B^T + \hat{C}_{BR} \end{array}\right)$ Loop-invariant 3	$C = \left(\begin{array}{c\|c} \hat{C}_{TL} & \star \\ \hline A_B A_T^T + \hat{C}_{BL} & A_B A_B^T + \hat{C}_{BR} \end{array}\right)$ Loop-invariant 4

Fig. 2. Loop-invariants for computing SYRK.

—Performance results are given for an SMP system based on the Intel Itanium2 architecture.

Together, these insights further the state-of-the-art in this area.

Overview. The article is organized as follows: In Section 2 we discuss the SYRK operation, four algorithmic variants for computing it, and the implementation of those algorithms using FLAME/C. The parallelization of the resulting implementations using OpenMP task queues and a custom workqueuing solution (referred to as "FLAME workqueuing") is discussed in Sections 3 and 4. Various issues related to load-balancing and 1D/2D partitioning are discussed in Section 5. Performance experiments are discussed and analyzed in Section 6. Concluding remarks are given in the final section.

2. A CONCRETE EXAMPLE

Consider the computation $C := AA^T + C$ where C is symmetric and only the lower triangular part of C is stored and updated. This operation is known as a *symmetric rank-k update* (SYRK).

The FLAME methodology describes how to derive linear algebra algorithms from predicates called *loop-invariants*, which describe intermediate states of the operation [Bientinesi et al. 2005]. Low et al. [2005] discuss how to arrive at four loop-invariants for the SYRK operation. These loop-invariants are given in Figure 2.[1]

For each loop-invariant, the FLAME methodology yields a corresponding algorithmic *variant*. Specifically, Loop-invariant i in Figure 2 yields algorithmic Variant i in Figure 3.[2] The loop-body of each algorithm contains two subproblems: a SYRK operation and a GEMM operation, each of which operates on smaller submatrices of A and C.

Having the ability to derive correct algorithms solves only part of the problem since translating those algorithms to code ordinarily requires delicate indexing into arrays, which exposes opportunities for the introduction of errors. We now illustrate how appropriately defined APIs overcome this problem [Bientinesi et al. 2005]. In Figure 4, we show an example of FLAME/C code corresponding

[1]The subscript notation used in Figure 2 simply identifies subpartitions of the matrices. This notation is shown in fuller context in Figure 3 and is further described in Bientinesi et al. [2005].
[2]A fifth loop-invariant and corresponding algorithmic variant exist for computing SYRK. Early work in this area found that the parallelization of this variant inherently requires heavy synchronization, which significantly limits speedup [Low et al. 2004]. We omit this variant from our discussion due to space constraints.

Algorithm: $C := \text{SYRK\_BLK\_VAR1\_2}(\ A,\ C\ )$

Partition $C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)$, $A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$

where C_{TL} is 0×0, A_T has 0 rows

while $m(A_T) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

where C_{11} is $b \times b$, A_1 has b rows

Variant 1:	**Variant 2:**
$C_{10} := A_1 A_0^T + C_{10}$	$C_{21} := A_2 A_1^T + C_{21}$
$C_{11} := A_1 A_1^T + C_{11}$	$C_{11} := A_1 A_1^T + C_{11}$

Continue with

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

endwhile

Algorithm: $C := \text{SYRK\_BLK\_VAR3\_4}(\ A,\ C\ )$

Partition $C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)$, $A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$

where C_{BR} is 0×0, A_B has 0 rows

while $m(A_B) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

where C_{11} is $b \times b$, A_1 has b rows

Variant 3:	**Variant 4:**
$C_{21} := A_2 A_1^T + C_{21}$	$C_{10} := A_1 A_0^T + C_{10}$
$C_{11} := A_1 A_1^T + C_{11}$	$C_{11} := A_1 A_1^T + C_{11}$

Continue with

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

endwhile

Fig. 3. Blocked algorithms for computing $C := AA^T + C$. The top algorithm implements Variants 1 and 2, corresponding to Loop-Invariants 1 and 2 in Figure 2. The bottom algorithm implements Variants 3 and 4, corresponding to Loop-Invariants 3 and 4 in Figure 2. Variants 1 and 2 share the same loop-body updates as Variants 4 and 3, respectively. Variants 1 and 2 sweep through C from the top-left to the bottom-right, and A from top to bottom, while Variants 3 and 4 traverse the matrices in the opposite directions.

```
1  FLA_Error Syrk_blk_var1( FLA_Obj A, FLA_Obj C, int nb_alg )
2  {
3    FLA_Obj CTL,   CTR,      C00, C01, C02,       AT,      A0,
4            CBL,   CBR,      C10, C11, C12,       AB,      A1,
5                             C20, C21, C22,                A2;
6    int b;
7
8    FLA_Part_2x2( C,    &CTL, &CTR,
9                         &CBL, &CBR,     0, 0, FLA_TL );
10   FLA_Part_2x1( A,    &AT,
11                        &AB,            0, FLA_TOP );
12
13   while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){
14     b = min( FLA_Obj_length( AB ), nb_alg );
15
16     FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,   &C00, /**/ &C01, &C02,
17                           /*************/  /*********************/
18                                             &C10, /**/ &C11, &C12,
19                            CBL, /**/ CBR,   &C20, /**/ &C21, &C22,
20                            b, b, FLA_BR );
21     FLA_Repart_2x1_to_3x1( AT,                &A0,
22                          /* ** */           /* ** */
23                                               &A1,
24                            AB,                &A2,   b, FLA_BOTTOM );
25     /*--------------------------------------------------------------*/
26     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
27               FLA_ONE, A1, A0, FLA_ONE, C10 );
28     FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
29               FLA_ONE, A1, FLA_ONE, C11 );
30     /*--------------------------------------------------------------*/
31     FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,  C00, C01, /**/ C02,
32                                                 C10, C11, /**/ C12,
33                               /**************/ /******************/
34                               &CBL, /**/ &CBR,  C20, C21, /**/ C22,
35                               FLA_TL );
36     FLA_Cont_with_3x1_to_2x1( &AT,                 A0,
37                                                    A1,
38                             /* ** */            /* ** */
39                               &AB,                 A2,     FLA_TOP );
40   }
41   return FLA_SUCCESS;
42 }
```

Fig. 4. FLAME/C code for a blocked implementation of Variant 1.

to Variant 1 in Figure 3. To understand the code, it suffices to know that `A` and `C` are descriptors for the matrices A and C, respectively. The various routines facilitate the creation of *views* into the data described by `A` and `C`. Think of a variable like `CTL` as a fancy pointer into the array corresponding to matrix C. Furthermore, the calls to `FLA_Gemm` and `FLA_Syrk` perform the same operations as the BLAS calls `dgemm` (matrix-matrix multiplication) and `dsyrk` (symmetric rank-k update). The most attractive feature of this code is the complete absence of loop and array indexing.

3. WORKQUEUING VIA PROPOSED OPENMP `TASKQ` AND `TASK` DIRECTIVES

OpenMP is a set of compiler directives and library routines that facilitate parallel programming on shared memory systems by allowing a programmer to explicitly specify regions of code that can be executed by simultaneous threads of execution [OpenMP Architecture Review Board 2006].

Shah et al. [1999] point out limitations of the primary constructs for creating OpenMP parallelism: the `parallel for` and `sections` directives. They note that the number of iterations in OpenMP `for` loops must be computable upon first entering the loop, precluding its use in many applications, such as traversing a linked-list of unknown length. Similarly, a `sections` construct containing n independent regions of computation, each marked by a `section` directive, is limited to achieving n-way parallelism [Shah et al. 1999]. The workqueuing model was proposed specifically to overcome these limitations.

3.1 The `taskq` and `task` Pragmas

A proposed OpenMP instantiation of the workqueuing model consists of two new directives: `taskq` and `task`. Conceptually, encountering a `taskq` directive causes the main thread to create an empty workqueue (or task queue). The code within the `taskq` scope is executed sequentially. As `task` directives are encountered, the code associated with the `task` block is encapsulated and enqueued as a unit of work onto the task queue. A number of other threads begin dequeuing and executing tasks from the queue according to a first-in/first-out (FIFO) scheduling policy. The main thread joins the others in processing tasks as soon as enqueuing is complete. When all tasks have been completed, the threads synchronize at the end of the `taskq` scope and continue through the program.

This OpenMP instantiation of the workqueuing model features two noteworthy properties:

—Workqueuing is dynamic, unlike the `sections` directive, which lexically encodes the degree of parallelism into the source code at compile time.

—Workqueuing is *flexible*, unlike the `parallel for` directive, which provides parallelism only for indexed `for` loops and also requires the number of instantiated work-shared task units to be computable at runtime.

These two properties of OpenMP workqueuing enable an attractive new mechanism for expressing parallelism within FLAME/C algorithm implementations.

3.2 Parallelization of SYRK

In Figure 5 we show how the `while` loop in Figure 4 can be annotated with OpenMP directives to create parallel tasks via the task queue mechanism. In Figure 5:

—Line 13 establishes the `taskq` block.

—Line 28 starts a section of code that defines a task to be added to the task queue. A single thread executes the `while` loop, enqueuing tasks as they are

```
13  #pragma intel omp parallel taskq
14  {
15    while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){
16      b = min( FLA_Obj_length( AB ), nb_alg );
17
18      FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,   &C00, /**/ &C01, &C02,
19                            /*************/  /**********************/
20                                               &C10, /**/ &C11, &C12,
21                             CBL, /**/ CBR,    &C20, /**/ &C21, &C22,
22                             b, b, FLA_BR );
23      FLA_Repart_2x1_to_3x1( AT,                &A0,
24                           /* ** */           /* ** */
25                                               &A1,
26                             AB,               &A2,   b, FLA_BOTTOM );
27      /*-----------------------------------------------------------*/
28      #pragma intel omp task captureprivate( A0, A1, C10, C11 )
29      {
30        FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
31                  FLA_ONE, A0, A1, FLA_ONE, C10 );
32        FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
33                  FLA_ONE, A1, FLA_ONE, C11 );
34      } /* end task */
35      /*-----------------------------------------------------------*/
36      FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,  C00, C01, /**/ C02,
37                                                  C10, C11, /**/ C12,
38                                /*************/ /******************/
39                                &CBL, /**/ &CBR,  C20, C21, /**/ C22,
40                                FLA_TL );
41      FLA_Cont_with_3x1_to_2x1( &AT,                 A0,
42                                                     A1,
43                              /* ** */            /* ** */
44                                &AB,                 A2,     FLA_TOP );
45    }
46  }   /* end of taskq */
```

Fig. 5. FLAME/C code from Figure 4 parallelized using OpenMP task queue directives.

encountered. The descriptors `A0`, `A1`, `C10`, and `C11` change with each iteration of the loop. The values of these descriptors must be captured at the time each task is enqueued so that the thread that dequeues the task will have the correct values to pass along to `FLA_Gemm` and `FLA_Syrk`.

—Line 34 ends the scope of the task being added to the queue.

—Line 46 ends the scope of the `taskq` block. Here, all threads are synchronized.

Clearly, task queues provide a simple mechanism for directing the parallel execution in this code.

3.3 Options

In Figure 5 the subproblems corresponding to the calls to `FLA_Gemm` and `FLA_Syrk` are independent and therefore can be executed in any order and/or queued as separate tasks. This is apparent by inspecting the algorithm itself. However, Low et al. [2005] discuss how to systematically detect the presence of independent loop iterations by inspecting the loop-invariants of the SYRK operation.

Furthermore, we may observe that the two updates within the loop-body are independent of one another within a single iteration. Given these two observations, we may modify our original parallelization shown in Figure 5 as follows.

One option is to split the single task in the loop-body of Figure 5 into two tasks:

```
#pragma intel omp task captureprivate( A0, A1, C10 )
{
  FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
            ONE, A1, A0, ONE, C10 );
}
#pragma intel omp task captureprivate( A1, C11 )
{
  FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
            ONE, A1, ONE, C11 );
}
```

A further observation is that the computations $C_{10} := A_1 A_0^T + C_{10}$ and $C_{11} := A_1 A_1^T + C_{11}$ (updating the lower triangle only) cost about $2bn(C_{10})n(A)$ and $b^2 n(A)$ floating-point arithmetic operations (FLOPs), respectively. Here $n(X)$ indicates the column dimension of matrix X. Notice that $n(C_{11})$ remains constant across iterations. Thus, the number of FLOPs required to compute the update to C_{11} is fixed. In contrast, $n(C_{10})$ grows linearly with each iteration of the loop and therefore the number of FLOPs required to update C_{10} increases proportionally as the algorithm iterates. This is unfortunate since costly tasks at the end of a scheduling queue can create a large load imbalance, as we will show later in Figure 7.

One way to overcome this problem is to execute the loop in reverse order (in compiler terms: apply a *loop reversal* transformation), since this would create the more costly tasks first. Variants 4 and 3 in Figure 3 execute the loops in Variants 1 and 2 in reverse, respectively.[3] In fact, Variants 1 and 3 have the property that tasks become more costly as the loop proceeds, while Variants 2 and 4 generate progressively less costly tasks. We will later show that differences in performance can be observed for different variants.

An alternative option replaces the single loop Figure 5 with two loops (in compiler terms: apply a *loop fission* transformation): the first loop for enqueuing the tasks that update C_{10}, and the second for enqueuing tasks that update C_{11}, as illustrated in Figure 6. The updates to C_{11} incur a smaller cost and result in fixed-sized tasks, compared to the updates of C_{10}, which result in larger, variable-sized tasks to be enqueued. These smaller tasks help balance the workload among threads before the threads synchronize at the end of the `taskq` block.

3.4 An Illustration of the Benefits of Different Options

The expected differences in performance are illustrated for Variants 2 and 3 in Figure 7. (Recall that Variants 2 and 3 are identical except that their loops

[3]This illustrates the value of the FLAME methodology, which can systematically find algorithmic variants that have different strengths and weaknesses.

```
#pragma intel omp parallel taskq
{
  while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) )
  {
    [ ... ]
    #pragma intel omp task captureprivate(A0, A1, C10)
    {
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
                ONE, A1, A0, ONE, C10 );
    }
    [ ... ]
  } /* end of first while loop */

  while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) )
  {
    [ ... ]
    #pragma intel omp task captureprivate(A1, C11)
    {
      FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                ONE, A1, ONE, C11 );
    }
    [ ... ]
  } /* end of second while loop */
} /* end of taskq */
```

Fig. 6. Outline of how the loop in Figure 5 can be implemented as two loops.

iterate in opposite directions.) In this figure, we simulate the scheduling of tasks to four threads for the different options described previously, where matrix C is 2400×2400, A is 2400×200, and the algorithmic blocksize b in Figure 3 equals 200 (except possibly during the last iteration). Each of the tasks is represented by a box that has a height proportional to the number of FLOPs required to complete the task. The y-axis itself is represented in units of millions of FLOPs. The integers in the boxes indicate the order in which the tasks are enqueued in the task queue.

The simulation results shown in Figure 7 were built upon a simplified workqueuing model that makes the following assumptions:

—Threads are idle when the computation begins.
—Enqueuing is instantaneous.
—When processing tasks, each thread computes at the same rate.
—Upon completing a task, the thread in question will dequeue a new task instantaneously. If the queue is empty, the thread becomes idle.
—The computation completes when the queue is empty and all threads are idle.

We see that Variant 2 in general performs better than Variant 3 since the cost of each variable-sized task decreases towards later iterations,allowing work to be more easily balanced among the threads before synchronization.Splitting the task in the loop-body into two tasks improves the load-balance for Variant 2, but not for Variant 3. Variant 2 achieves near-perfect load-balance by splitting the loop into two loops, with variable-sized tasks enqueued in the first loop while the smaller fixed-sized tasks are enqueued by the second. This change

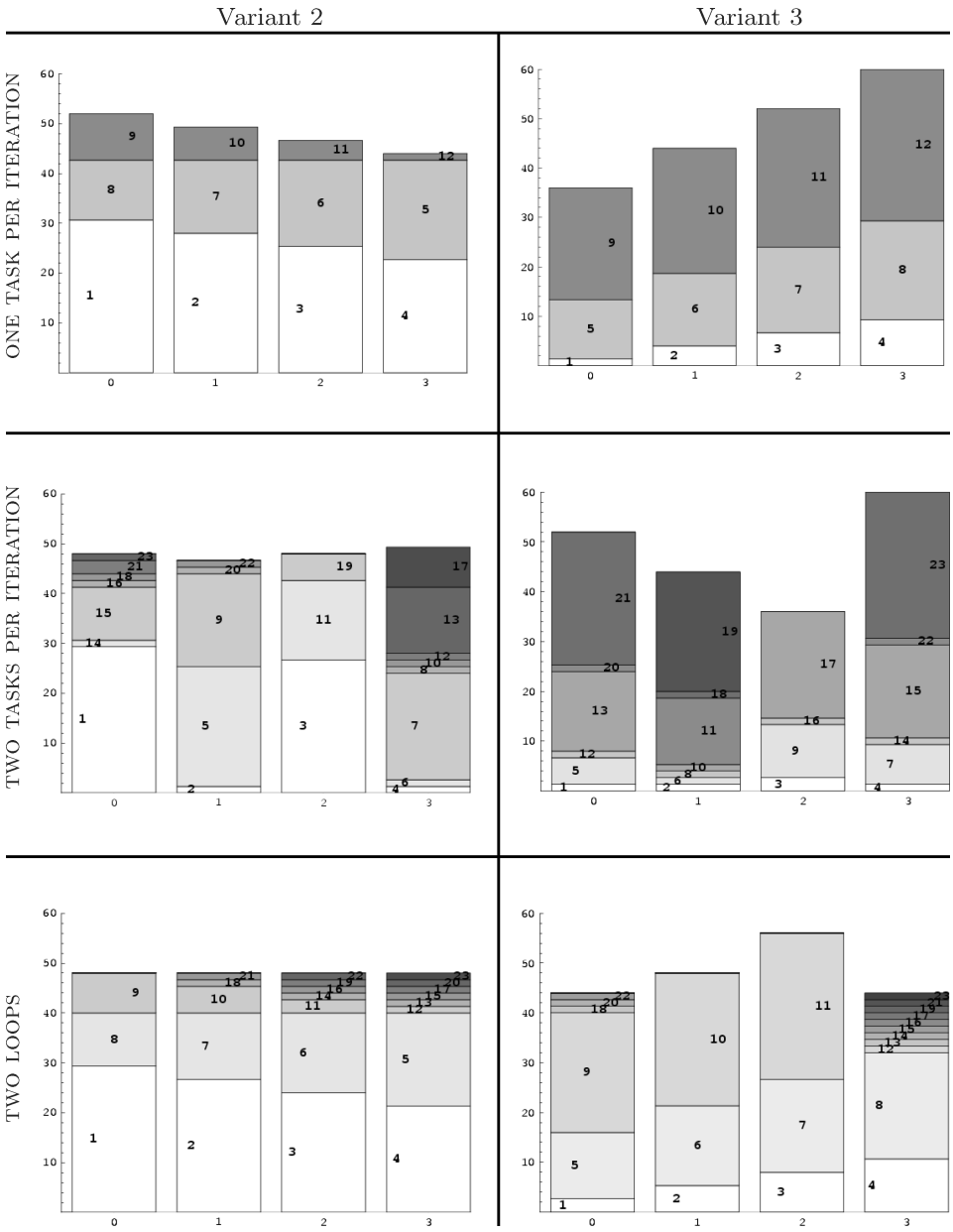


Fig. 7. Simulated OpenMP task queues scheduling of tasks to four threads for Variants 2 and 3 when $m = 2400$, $k = 200$, and the blocksize equals 200. The sum of the heights of the rectangles in each x-axis column correspond to the total amount of work assigned to each thread. The y-axis is in units of FLOPs $\times 10^6$ (millions of FLOPs). Load balance is ideal when all threads receive equal work. *Bottom line:* Load balance is determined by the number of tasks created, the cost of each task, and the order in which tasks are enqueued.

provides only a modest improvement to Variant 3; the imbalance created by the increasing cost of variable-sized tasks is simply too large to erase with the smaller subproblems.

4. AN ALTERNATE IMPLEMENTATION: FLAME WORKQUEUING

As of this writing, the proposed OpenMP task queuing mechanism has two short comings. First, task queues are an experimental implementation: to our knowledge it is currently only supported by the Intel compiler [Su et al. 2002]. The current official OpenMP specification (version 2.5) does not provide any workqueuing constructs.[4] Second, our discussion in Section 3.4 suggests that the OpenMP task scheduling does not always result in good load-balance among threads. Later in this section, we discuss a custom implementation for FLAME that addresses these shortcomings.

4.1 Theoretical Basis for Dequeuing Tasks Largest to Smallest

The problem of scheduling tasks among a set of threads is a variation of the Bin-Packing Problem in which a collection of objects are packed into a set of bins such that the total weight or volume of each bin does not exceed a given threshold [Weisstein 2006]. In our case of workqueuing, the computational tasks correspond to objects being packed (or scheduled) while the task volumes correspond to their computational costs, which can be approximated by counting FLOPs. Threads correspond to the bins into which the objects are packed. The target bin capacity corresponds to the sum of the tasks to be executed divided by the number of threads—that is, the ideal amount of computation per thread. J. D. Ullman [Garey et al. 1973] proves that a naive algorithm, one that packs objects into the first available bin with space, is suboptimal by as much as 70% [Weisstein 2006]. Johnson [1973] shows that an algorithm that first sorts the objects from largest to smallest will be at most 22% suboptimal [Weisstein 2006]. This suggests that, in general, an execution of tasks scheduled from largest to smallest with respect to computational cost will always perform reasonably well compared to executions based upon other task schedulings.

4.2 Motivation for Sorting Tasks Over Changing the Algorithm

Ideally, an application loop will create independent subproblems of equal cost. Such cases usually parallelize well with minimal effort. Other loops may naturally create subproblems that decrease monotonically in cost. Given the discussion in the previous section, this is the next best scenario if subproblems of equal cost are not possible. If the loop creates subproblems that increase monotonically in cost, then a simple loop reversal will cause the tasks to be enqueued in the desired order. As we saw in Figure 7, sometimes even more changes are needed, as the best scheduling also required splitting the main algorithm loop into two separate loops to enqueue variable-sized tasks first, followed by smaller fixed-sized tasks. But this method of changing the algorithm to induce a desirable scheduling is suboptimal for two reasons. First,

[4]It is possible that workqueuing support will be added to version 3.0 of the OpenMP specification.

it requires nontrivial changes to the algorithm code. Uniquely, FLAME codes resemble the underlying algorithm so closely that changing the source code will tend to also obscure the algorithm itself. More generally, changing the algorithm code is also less than desirable from a code maintenance perspective. Second, it is possible that there does not exist a reasonable loop or code transformation for a given algorithm that enqueues tasks from largest to smallest. For example, it is conceivable that some loops may create subproblems whose cost neither increases nor decreases monotonically.

An alternate way to ensure a desirable ordering of tasks within the queue, one that is less disruptive to the algorithm and therefore more portable, is to allow the application loop to enqueue tasks normally and then sort the queue before parallel execution. This solution is quite flexible and its disadvantages are minor.[5]

Unfortunately, the OpenMP `taskq` construct as specified provides no such sorting mechanism [Shah et al. 1999; Su et al. 2002]. The responsibility for ensuring load-balancing through a desirable task ordering is left to the programmer. Furthermore, experience, as well as results discussed later in this article, show that the performance penalty for executing tasks from an unsorted queue can be quite severe.

4.3 FLAME Workqueuing

To circumvent the shortcomings of OpenMP task queues, we have implemented a custom workqueuing solution that behaves much like task queues, although domain-specific to FLAME, featuring the following enhancements:

—Portability. Our implementation does not use the `taskq` or `task` constructs and thus works on any conventional implementation of OpenMP. In fact, FLAME workqueuing abstracts all implementation details from the API, potentially allowing us to replace OpenMP altogether with a more portable threading mechanism such as POSIX threads.

—Task sorting. FLAME workqueuing automatically sorts the task queue according to each task's estimated cost (FLOP count) before parallel execution begins.

A programmer enqueues a routine by replacing it with a corresponding preprocessor macro. The macro inserts an invocation to `FLA_Queue_push()`, which uses the data associated with the original function call to create a task structure. This structure is added to the queue, which is implemented as a linked list. After enqueuing is complete, the programmer signals that execution may begin by calling `FLA_Queue_exec()`, the definition of which is shown in Figure 8. The linked list of tasks is indexed and then sorted according to approximate FLOP cost using Quicksort. Finally, the sorted queue's tasks are executed in parallel using a `parallel for` directive with dynamic scheduling.

[5]Threads must wait for sorting to finish before parallel execution of the queue can begin. The sorting itself is an $O(nlog(n))$ operation that will most likely not adversely affect performance for our applications.

```
1  void FLA_Queue_exec()
2  {
3    int        i, n_tasks;
4    FLA_Task** task_array;
5    FLA_Task*  t;
6
7    /* The queue is full, so we may now create an index of each task. */
8    FLA_queue_create_task_array();
9
10   /* Sort the task_array with standard C library function qsort(). */
11   FLA_queue_sort_task_array();
12
13   /* Copy the task_array pointer and n_tasks integer locally to
14      satisfy the OpenMP compiler. */
15   n_tasks    = task_queue.n_tasks;
16   task_array = task_queue.task_array;
17
18   /* Iterate over the task queue using the random-access task_array.
19      Note that we iterate backwards because qsort() sorts in ascending
20      order, while we want to execute the tasks in descending order. */
21   #pragma omp parallel for shared( task_array, n_tasks ) \
22                            private( i, t ) \
23                            schedule( dynamic, 1 )
24   for( i = n_tasks - 1; i >= 0; --i )
25   {
26     t = task_array[ i ];
27     FLA_queue_exec_task( t );
28   }
29
30   /* Flush the queue: walk the task_array and free() each element. */
31   FLA_queue_flush();
32
33   /* Free the task_array now that all tasks have been executed. */
34   FLA_queue_free_task_array();
35 }
```

Fig. 8. Code fragment from the FLAME workqueuing implementation: definition of `FLA_Queue_exec()`.

The reader should note that FLAME workqueuing was not intended to replace the functionality of OpenMP task queues. Clearly the implementors of task queues are providing a generalized solution that leverages the privileged access that the compiler has to the code before compilation. Our workqueuing implementation was designed primarily to minimize disturbance to conventional algorithms implemented with the FLAME/C API while providing a mechanism to sort the contents of the queue prior to execution.

Though they are motivated by the same conceptual model, the FLAME workqueuing API uses a different syntax than the task queue constructs. Figure 9 shows how subproblems are enqueued as tasks under the FLAME workqueuing API. The programmer must initialize the workqueuing environment by invoking `FLA_Queue_init` and calling `FLA_Queue_finalize` to free internal resources when workqueuing is no longer needed. The API contains no direct analog to the `taskq` directive. Consequently, nested queuing is not

```
13  FLA_Queue_init();
14  [ ... ]
15  while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){
16    b = min( FLA_Obj_length( AB ), nb_alg );
17
18    FLA_Repart_2x1_to_3x1( AT,                &A0,
19                         /* ** */            /* ** */
20                                               &A1,
21                           AB,                 &A2,        b, FLA_BOTTOM );
22    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,       &C00, /**/ &C01, &C02,
23                         /* ************* */   /* ******************** */
24                                                 &C10, /**/ &C11, &C12,
25                           CBL, /**/ CBR,        &C20, /**/ &C21, &C22,
26                           b, b, FLA_BR );
27    /*-----------------------------------------------------------*/
28    ENQUEUE_FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
29                      FLA_ONE, A1, A0, FLA_ONE, C10 );
30    ENQUEUE_FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
31                      FLA_ONE, A1, FLA_ONE, C11 );
32    /*-----------------------------------------------------------*/
33    FLA_Cont_with_3x1_to_2x1( &AT,                 A0,
34                                                   A1,
35                            /* ** */             /* ** */
36                              &AB,                 A2,     FLA_TOP );
37    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,       C00, C01, /**/ C02,
38                                                     C10, C11, /**/ C12,
39                            /* ************** */  /* ****************** */
40                              &CBL, /**/ &CBR,       C20, C21, /**/ C22,
41                              FLA_TL );
42  }
43  [ ... ]
44  FLA_Queue_exec();
45  [ ... ]
46  FLA_Queue_finalize();
```

Fig. 9. FLAME/C code from Figure 4 parallelized using FLAME workqueuing.

supported and all tasks are implicitly enqueued onto the same global queue. Also, whereas OpenMP task queues automatically dispatch threads as soon as the first task is enqueued, the programmer of the FLAME workqueuing API must explicitly invoke parallel execution after enqueuing is complete, by calling `FLA_Queue_exec`.[6] Presumably, this may cause FLAME workqueuing to slightly underperform a similar code that uses OpenMP task queues when the algorithm happens to enqueue tasks in the desired sorted order. However, performance results in Section 6.1 show that this penalty is negligible given the relatively small number of tasks enqueued when executing a parallel SYRK operation.

[6]In order to guarantee a fully sorted queue, the programmer must allow enqueuing to finish. While dispatching threads immediately may sound desirable, it denies the workqueuing implementation the opportunity to sort the queue before execution. Figure 13 confirms that the benefits of deferring execution to allow sorting to take place dwarf the added serialization costs. Conceivably, there exist applications that create many small tasks for which this assertion may not hold. In that case, sorting is probably not necessary anyway due to the fact that many small tasks tend to inherently yield good load-balancing under a FIFO scheduling policy.

5. BLOCKING AND PARTITIONING

In our previous discussions of the parallel workqueuing codes shown in Figures 5 and 9 we did not mention a subtle but important detail: How does one determine the algorithmic blocksize b? In the workqueuing-enabled SYRK implementations, this blocksize determines the dimensions of the subproblems created, which corresponds directly to the cost of the tasks placed onto the queue. Determining an appropriate blocksize leads us to a somewhat more general discussion of how best to partition the computation into tasks. In this section we describe some methods of partitioning submatrices that may help us attain better load-balancing among the threads.

5.1 Proportional Blocking

Notice that for Variant 2 in Figure 3, the cost in FLOPs of the variable-sized `FLA_Gemm` task created in the ith iteration decreases linearly with i while the cost of the `FLA_Syrk` task remains constant and relatively small across all iterations. Furthermore, the number of tasks generated varies with the m dimension. Naively, we may choose a blocksize a priori without regard to the problem size. We refer to this method of choosing b as an arbitrary fixed value as *constant* blocking. However, the simulation in Figure 10 reveals that this method results in load-imbalance and diminished parallel performance for certain smaller matrix dimensions.

In order to circumvent this problem, let us chose the algorithmic blocksize b so that the algorithm cycles through $2t$ iterations, regardless of the problem size determined by the m dimension. Under this scenario, the algorithm creates $2t - 1$ variable-sized tasks and $2t$ fixed-sized tasks, where t equals the number of processor-bound[7] threads participating in the computation. Under our simplified task scheduling model, this careful choice of blocksize always yields an ideal load balance similar to the scheduling shown in the bottom-left graph of Figure 10. Therefore, a good value for b may be chosen to equal $\frac{m}{2t}$. We refer to this method of choosing b as a function of the partitioned dimension m and number of threads t as *proportional* blocking.[8]

Notice, however, that `dgemm` executes more efficiently the larger b is chosen. Thus, for SYRK problems with small dimensions, we expect proportional blocking will yield better load-balance but possibly at the expense of worse performance by each thread.

[7]The operating system should, either by default or at the behest of the programmer, "bind" each thread to a single unique processor to ensure optimal performance. Under version 2.6 of the Linux kernel, programmers may explicitly request this behavior by setting the thread's CPU *affinity* via the `sched_setaffinity` function [Dow 2005]. Experience suggests that setting the CPU affinity of threads in a parallel computation is typically desirable. Otherwise, the scheduler may direct threads to migrate between processors. This situation may lead to significant performance degradation as a recently migrated thread may experience a higher latency while accessing data resident on the cache or local memory associated with its previous CPU.

[8]Given that $b = \frac{m}{2t}$, it follows that $m = 2bt$. The former equation suggests how to choose, as a function of the problem size, the largest possible b that still induces ideal load-balance. The latter equation reveals the smallest problem size m for which constant blocking will yield peak load-balance. The bottom-left graph in Figure 10 illustrates both of these scenarios.

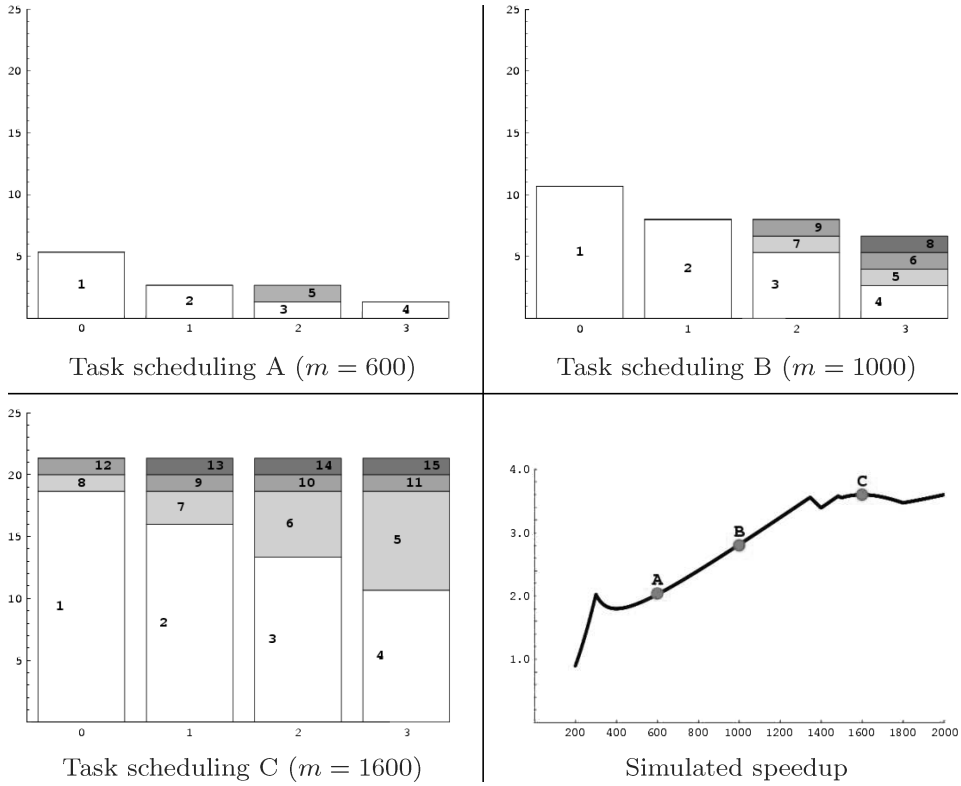


Fig. 10. Simulated OpenMP task queues scheduling of tasks to four threads for Variant 2 shown for select values of m when $k = 200$. The two-loop parallelization for Variant 2 was used, which enqueues tasks in sorted order. Also, the blocksize used was 200. The axes' units of the three task scheduling graphs is similar to that of Figure 7. The corresponding speedup of each task scheduling is marked on the curve shown in the lower-right graph, in which we assume `dgemm` and `dsyrk` consistently perform at 90% of peak efficiency. Bottom line: Given a one-dimensional constant blocking, some threads fall idle before others due to poor distribution of work, where the largest variable-sized GEMM tasks become bottlenecks. This load-imbalance hinders parallel speedup and thus results in limited performance for many smaller problem sizes.

5.2 Partitioning in Two Dimensions

So far, we have only considered a single blocksize b in the SYRK algorithms shown in Figure 3. Recall that this blocksize determines two properties of the tasks enqueued during each iteration. For Variant 2, these two properties are the dimension (order) of C_{11}, which is updated in the `FLA_Syrk` subproblem, and the dimensions of the C_{21} panel that is updated by the `FLA_Gemm` subproblem. The C_{11} submatrix is already small and creates a task of constant cost. However, the C_{21} submatrix varies in size and provides us with the opportunity to further partition along its m (row) dimension. Let us consider an alternate version of Variant 2 in which we replace the call to `FLA_Gemm` with a GEMM variant that partitions A_2 and C_{21} along the m dimension with a blocksize that is independent of the value of b used thus far. Partitioning A_2 and C_{21} causes the the parallelized algorithm to enqueue a larger number of somewhat smaller

```
1  FLA_Error Gemm_blk_var1( FLA_Obj A, FLA_Obj B, FLA_Obj C, int nb_alg )
2  {
3    FLA_Obj AT,        A0,        CT,        C0,
4            AB,        A1,        CB,        C1,
5                       A2,                   C2;
6    int b;
7
8    FLA_Part_2x1( A,    &AT,
9                        &AB,            0, FLA_TOP );
10   FLA_Part_2x1( C,    &CT,
11                       &CB,            0, FLA_TOP );
12
13   while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){
14     b = FLA_Task_compute_blocksize( A, AT, FLA_TOP, nb_alg );
15
16     FLA_Repart_2x1_to_3x1( AT,                &A0,
17                          /* ** */            /* ** */
18                                               &A1,
19                            AB,                &A2,        b, FLA_BOTTOM );
20     FLA_Repart_2x1_to_3x1( CT,                &C0,
21                          /* ** */            /* ** */
22                                               &C1,
23                            CB,                &C2,        b, FLA_BOTTOM );
24     /*--------------------------------------------------------------*/
25     ENQUEUE_FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
26                       FLA_ONE, A1, B, FLA_ONE, C1 );
27     /*--------------------------------------------------------------*/
28     FLA_Cont_with_3x1_to_2x1( &AT,               A0,
29                                                  A1,
30                             /* ** */            /* ** */
31                               &AB,               A2,     FLA_TOP );
32     FLA_Cont_with_3x1_to_2x1( &CT,               C0,
33                                                  C1,
34                             /* ** */            /* ** */
35                               &CB,               C2,     FLA_TOP );
36   }
37   return FLA_SUCCESS;
38 }
```

Fig. 11. FLAME/C code, ready for use with FLAME workqueuing, implementing a variant of GEMM that transposes B and partitions A and C along the m dimension. This code may be called in our FLAME/C variant 2 of SYRK instead of calling `FLA_Gemm` directly.

variable-sized tasks for each iteration of the Variant 2 `while` loop. Figure 11 shows the FLAME/C code that implements this variant of GEMM. This effectively allows the workqueuing analog of a two-dimensional data decomposition. The advantages of this approach are twofold:

—First, attaining good load-balance is easier when enqueuing smaller partitioned `FLA_Gemm` tasks than when the subproblems are enqueued unpartitioned. This is simply a consequence of the fact that smaller tasks more easily allow a scheduling that distributes work equally across all threads. This behavior holds regardless of whether the task queue is sorted.

—Second, we may leverage proportional blocking so that the load-balance of a 2D partitioning remains ideal under our model. By using proportional blocking to partition A_2 and C_{21} along their m dimensions into q subpartitions of roughly equal size, we may choose the SYRK blocksize b to equal $\frac{mq}{2t}$ (resulting in $\frac{m}{b} = \frac{m}{(\frac{mq}{2t})} = \frac{2t}{q}$ iterations in the SYRK algorithm). This blocksize is larger than the value proposed in Section 5.1 and thus should allow the `dgemm` implementation to perform more efficiently, especially for smaller problems. However, this will also proportionally reduce the number of SYRK subproblem tasks from $2t$ to $\frac{2t}{q}$ and similarly increase their costs. This smaller number of more costly fixed-sized tasks may be more difficult to divide equally among t threads for $q > 2$.

We show later in Section 6.1 that a two-dimensional partitioning with proportional blocking allows us to sustain good performance per thread as well as good load-balance across threads for smaller problem sizes than would be possible with a one-dimensional partitioning.

6. EXPERIMENTS

Workqueuing comparison. To demonstrate the effect of algorithmic variants and task partitioning methods on performance, we parallelized each of the four variants using OpenMP task queues and FLAME workqueuing. In the case of the variants using OpenMP task queues, we applied the code transformation options described in Section 3.3 to arrive at three parallelized configurations: a simple insertion of the task queue directives with one task in the loop-body; the separation of the two updates to create two tasks in the loop-body; and the separation of the fixed- and variable-sized tasks into two separate loops. For FLAME workqueuing, we implemented Variants 1 through 4 and replaced the invocation of `FLA_Gemm` with a call to a suitable GEMM variant to enable further partitioning of the variable-sized tasks.

Blocking and Partitioning. Experiments were set up to accept constant or proportional algorithmic blocksizes in both one- and two-dimensional partitionings. Blocksizes are presented as follows: a blocksize *pair* $b = (x, y)$ indicates that a blocksize x is used in the overall SYRK algorithm while a blocksize y is used to further partition the GEMM subproblem. We denote constant blocksizes with positive numbers while proportional blocksizes are encoded as negative values. For negative x (or y), the algorithm completes $|x|$ (or $|y|$) iterations along the partitioned dimension where the actual blocksize value used is approximately equal for all iterations. If $y = -1$, then the algorithm does not partition the GEMM subproblem, which corresponds to an overall one-dimensional partitioning for the SYRK algorithm. Figure 12 illustrates the potential variations in task scheduling induced by one- and two-dimensional partitionings when combined with constant and proportional blockings. Also, we have extended the FLAME/C programming interface to include a function, `FLA_Task_compute_blocksize`, which computes the value of `b` for each iteration based on the blocksize pair values provided as input. This routine is used to compute blocksizes in all experiments. Its use is demonstrated in Figure 11.

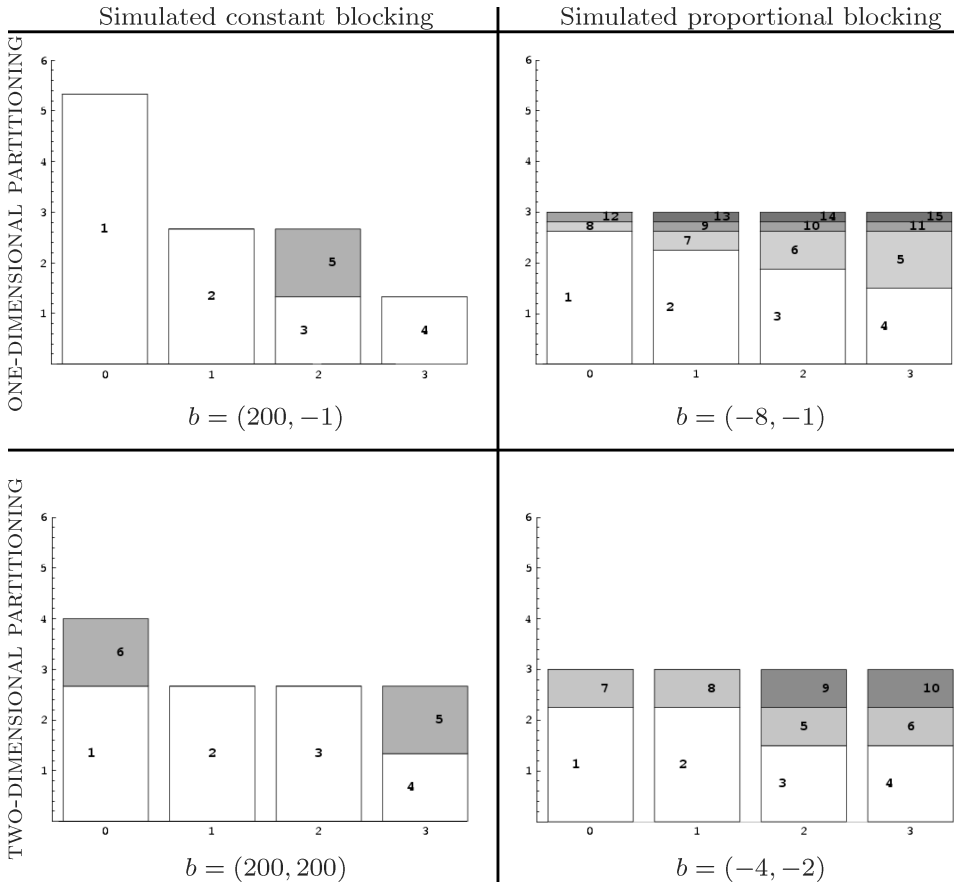


Fig. 12. Simulated scheduling of tasks to four threads for Variant 2 when $m = 600$ and $k = 200$ for various blocksize and partitioning schemes. The axes' units are similar to those of Figures 7 and 10. Bottom line: Depending on the problem size, tasks created with constant blocking sometimes fail to distribute well among threads. Moving to a constant 2D partitioning helps, but remains suboptimal. In constrast, both 1D and 2D proportional blocking create the opportunity for ideal load-balance regardless of problem size.

Software. The Intel C compiler, version 9.0, was used to compile source code, since it is the only major compiler to support the proposed OpenMP task queue extensions. Calls to `FLA_Gemm` and `FLA_Syrk` in the loop-body were defined as wrappers to implementations of the BLAS routines `dgemm` and `dsyrk`, respectively. The code was linked to a sequential build of the GotoBLAS library, version 1.07, by Kazushige Goto [Goto 2006]. Also, threads were bound to unique processors using the SGI `dplace` utility. This method is easier and less intrusive, though less portable, than using the `sched_setaffinity` routine present in the Linux kernel.

Hardware. Performance was measured on an SGI Altix ccNUMA server consisting of seven dual-processor Itanium2 compute nodes, or *bricks*. Each brick contains approximately 2GB of local physical memory, but logically

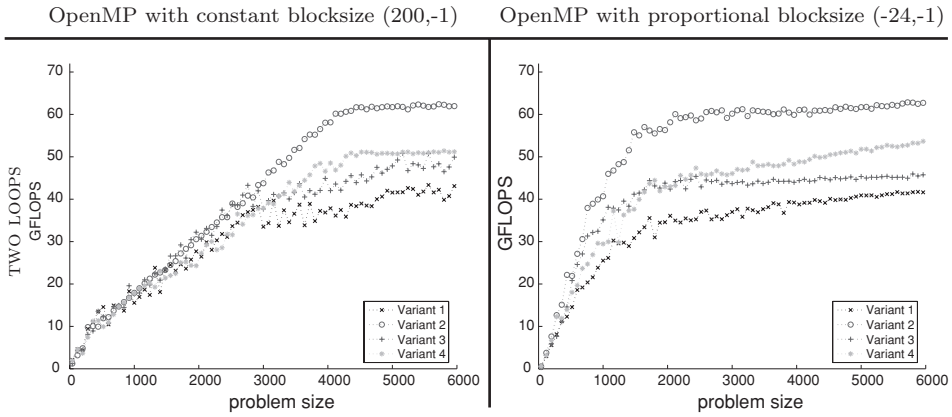


Fig. 13. Performance of OpenMP task queue parallelizations (12 threads) of SYRK Variants 1 through 4 where m equals the problem size and $k = 200$. The experiments on the left were performed with a constant blocksize of 200 while those on the right were performed with a proportional blocksize that partitioned the matrix equally through 24 iterations. Only results for the two-loop task partitioning option are shown. Bottom line: Proportional blocking enables superior load balancing for small to medium-sized problems, allowing performance to ramp up quickly. Variant 2 outperforms all other variants due to a combination of enqueuing variable-sized tasks in descending order of cost and properties inherent in the `dgemm` implementation used to compute the matrix products associated with these tasks.

shares its memory with all other nodes via SGI's NUMAflex shared-memory architecture. Each CPU is clocked at 1.5GHz and may execute up to four double precision floating-point operations per clock cycle, yielding a peak performance of 6 GFLOPS (10^9 FLOPs/sec.) per processor. Thus, the total peak performance of the system is 84 GFLOPS. However, while 14 processors were available, we limited our tests to using 12 threads. Therefore, the maximum attainable peak of our experiments is 72 GFLOPS.

Computations. All computations were performed in double precision (64 bit) floating-point arithmetic. For the purposes of computing the rate of computation, the SYRK operation count is m^2k FLOPs for $C \in \mathbb{R}^{m \times m}$ and $A \in \mathbb{R}^{m \times k}$. The GFLOPS rate reported in the graphs was computed by the formula

$$\text{GFLOPS attained} = \frac{m^2 k}{\text{time (in sec.)}} \times 10^{-9}.$$

6.1 Results

The resulting performance is reported in Figures 13–17. For graphs reporting absolute performance, the maximum of the y-axis is set to 72 GFLOPS to allow the reader to visually evaluate the results relative to the theoretical peak of the experiments.

OpenMP task queues with 1D partitioning. Figure 13 shows two graphs containing results for OpenMP task queue parallelizations of Variants 1

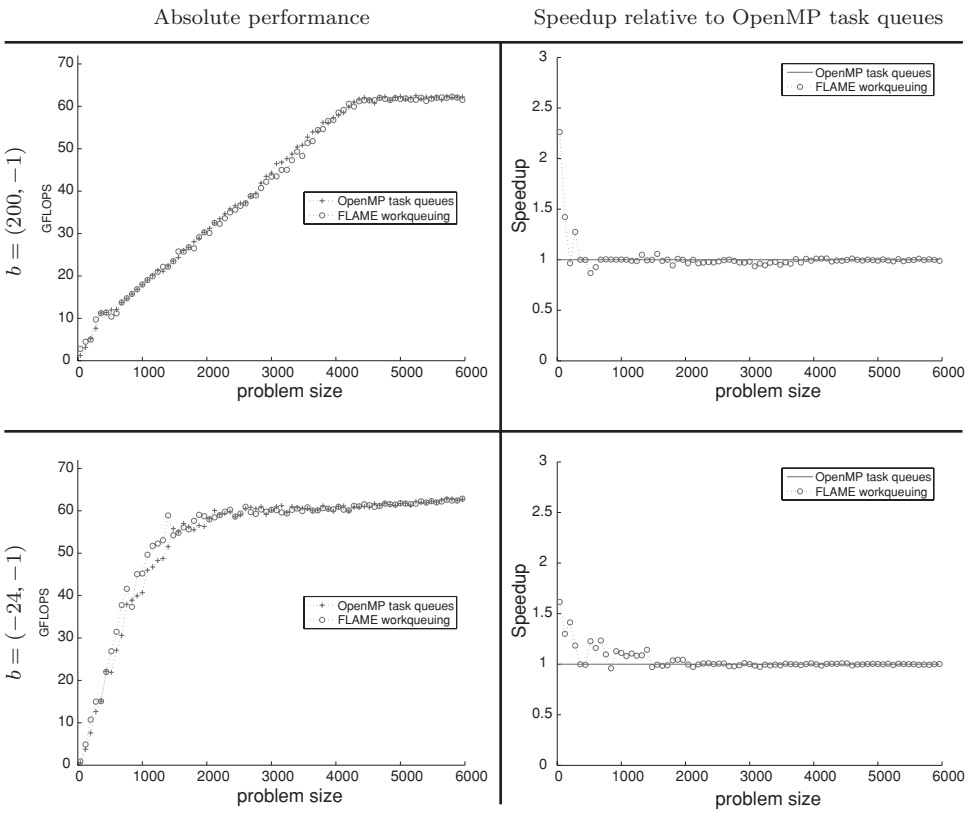


Fig. 14. Performance of OpenMP and FLAME workqueuing parallelizations (12 threads) of SYRK Variant 2 when m equals the problem size and $k = 200$. Absolute performance is shown in the left column for two one-dimensional partitionings while corresponding FLAME speedup relative to OpenMP is shown on the right. *Bottom line:* When compared to OpenMP task queues, FLAME workqueueing overhead is negligible if not nonexistent.

through 4. Results are shown for only the two loop task partitioning option.[9] The left and right graphs use a constant blocksize of 200 (ie: $b = (200, -1)$) and a proportional blocking of -24 (ie: $b = (-2t, -1) = (-24, -1)$), respectively. Both graphs show results from one-dimensional partitionings, as indicated by the GEMM blocksize of -1 in the blocksize pairs. These results clearly show that a constant blocksize of 200 is suboptimal for most problem sizes tested. Performance is greatly improved by partitioning with a proportional blocksize of -24. This observation holds regardless of how tasks are enqueued. Also of interest are the best and worst performing variants. Variants 2 and 4 both enqueue variable-sized tasks in descending (naturally sorted) order of cost. However, Variant 2 consistently outperforms Variant 4. This is likely due to the fact that the variable-sized GEMM tasks enqueued by Variant 2 update C_{21} with $A_2A_1^T$,

[9]We have omitted graphs for the other two options discussed in Section 3.3, as they exhibited performance signatures similar to those shown. The curious reader may find further discussion of all three task partitioning options in an earlier study of the topic presented in Low et al. [2004].

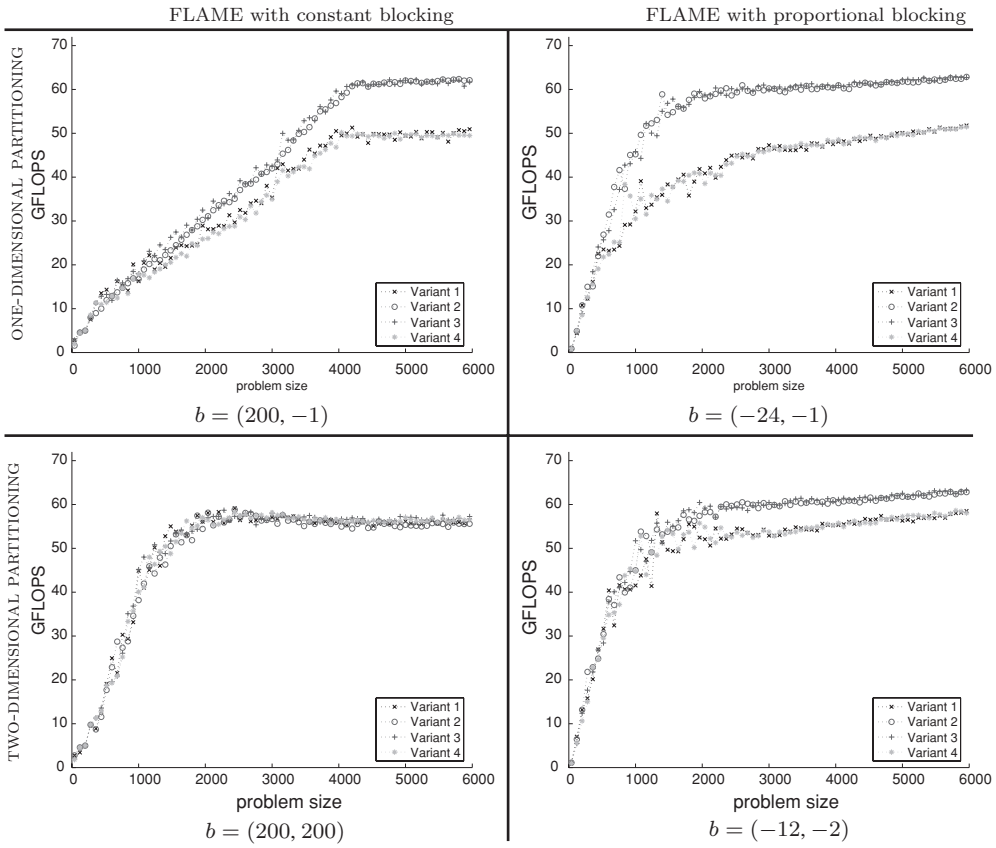


Fig. 15. Performance of FLAME workqueuing parallelizations (12 threads) of SYRK Variants 1 through 4 when m equals the problem size and $k = 200$. Bottom line: Sorting the workqueue renders Variants 1 and 2 computationally indistinguishable from Variants 4 and 3, respectively. Constant 2D partitioning performs better for many midsized problems but falls short of constant 1D performance for large problems due to limited `dgemm` efficiency on small 200×200 blocks. Proportional 2D partitioning performs similarly to that of proportional 1D for Variants 2 and 3; benefits appear mostly limited to improving lackluster Variants 1 and 4.

where C_{21} and A_2 are column-panel matrices. As of this writing, the sequential `dgemm` routine in GotoBLAS is more optimized for matrix multiplication on operands of this shape than the shape of operands in Variant 4, which updates C_{10} with $A_1 A_0^T$, where C_{10} and A_0^T are row-panel matrices.

OpenMP task queues v. FLAME workqueuing. Figure 14 shows the performance of SYRK Variant 2 using FLAME workqueuing and OpenMP task queues. The two graphs on the left show absolute performance while the graphs on the right show the speedup of FLAME workqueuing relative to OpenMP task queues. One set of graphs is given for each of the two blocksize pairs, $b = (200, -1)$ and $b = (-24, -1)$, used in Figure 13. These results demonstrate that FLAME workqueuing does not incur significant overhead compared to the OpenMP task queue implementation in the Intel compiler. In fact, for a

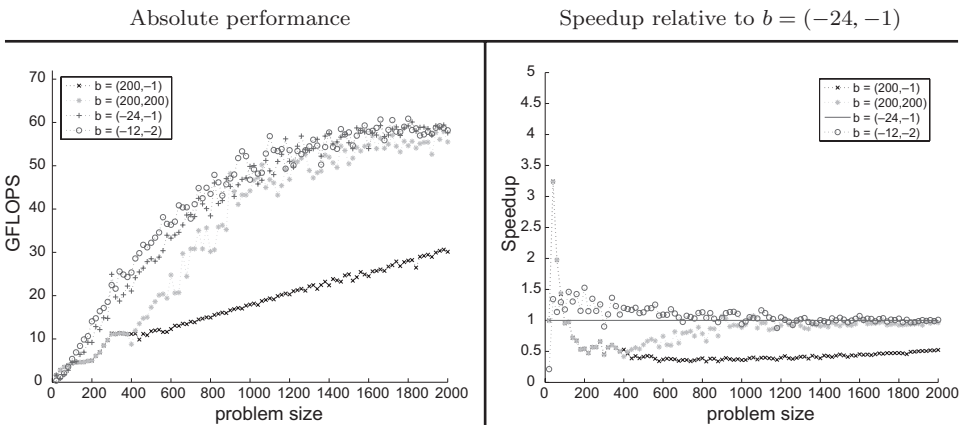


Fig. 16. Performance of FLAME workqueuing parallelization (12 threads) of SYRK Variant 2 when m equals the problem size and $k = 200$. The problem size range and increments have been reduced from Figure 13 to show more detail for small problems. Bottom line: A proportional 2D partitioning performs noticably better for certain smaller problems and on par with a proportional 1D partitioning for larger problems.

narrow range of small problems, FLAME workqueuing outperforms OpenMP task queues more often than not. Because the two implementations perform so similarly, we feel justified in limiting the remaining experiments to FLAME workqueuing.

Constant v. proportional blocking/1D v. 2D partitioning. In Figure 15, we highlight the differences among the four variants when constant and proportional blocksizes are used in one- and two-dimensional partitionings. For constant blocking, a blocksize of 200 is used. (We will see the effect of reducing this blocksize later.) For proportional two-dimensional partitioning, we chose $b = (-t, -2) = (-12, -2)$. This blocksize pair has the special property that it partitions the SYRK algorithm into a minimal number of iterations such that enough variable-sized tasks are produced to distribute well when $t = 12$, while still creating an equal number of fixed-sized tasks for all threads. The results in Figure 15 lead us to three interesting observations:

—Variants 1 and 4 perform nearly identically—likewise for Variants 2 and 3. The explanation is straightforward. Variants 1 and 2 share the same loop-body updates with Variants 4 and 3, respectively; the only difference within the algorithm pairs is the order in which subproblems are enqueued as tasks. FLAME workqueuing sorts the task queue automatically before threads begin dequeuing work, rendering Variants 1 and 2 equivalent to, and computationally indistinguishable from, Variants 4 and 3 respectively. In addition, constant 2D partitionings cause variable-sized GEMM subproblems to be broken almost entirely into homogeneous 200×200 tasks, rendering the performance signatures of *all four* variants identical.

—The results show an overall performance advantage for 2D partitionings when a constant blocksize is used. Similarly, moving from a constant 1D partitioning to one that uses proportional blocking is sufficient to see a large

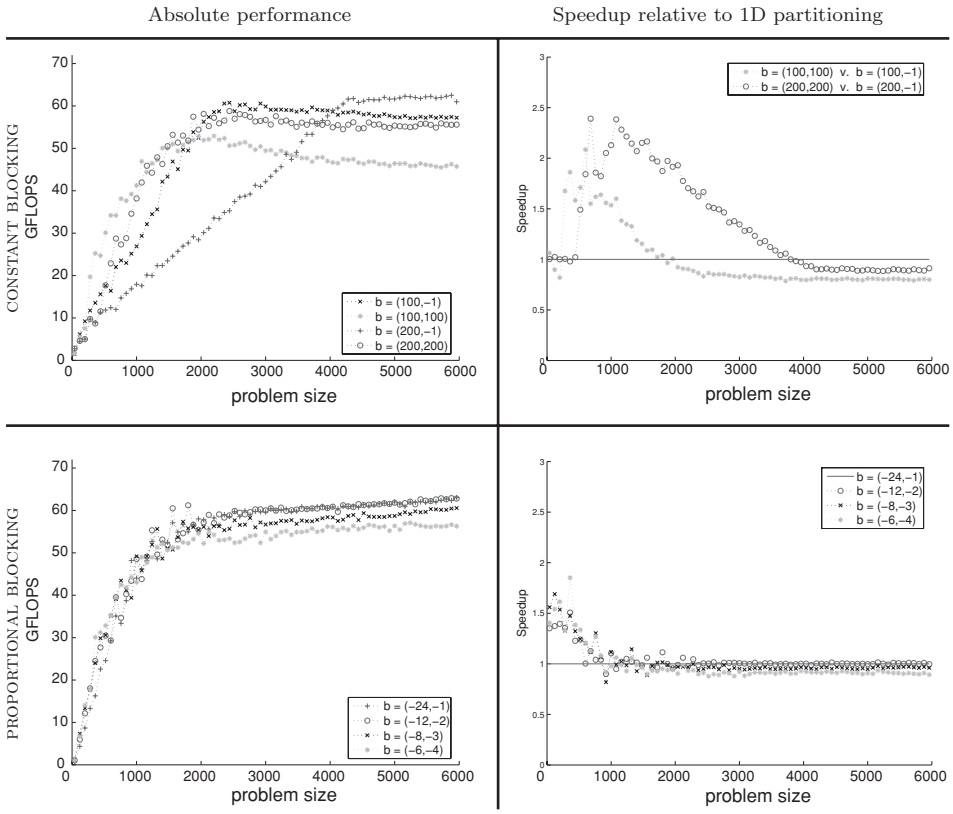


Fig. 17. Performance of FLAME workqueuing parallelizations (12 threads) of SYRK Variant 2 when m equals the problem size and $k = 200$. *Bottom line:* Two-dimensional partitioning is beneficial for smaller problems when compared to a similar 1D partitioning. A smaller constant blocksize generates parallelism more quickly, but at the expense of lower `dgemm` efficiency.

jump in performance for a wide range of problems. In fact, the graphs suggest that Variants 2 and 3 need not partition both proportionally and in two dimensions, but rather only proportionally, in order to attain high performance for a wide range of problem sizes. This observation is predicted by the simulation of proportional 1D and 2D partitionings reported in Figure 12.

—Last, it is interesting to note that a two-dimensional partitioning noticeably improves the performance of the otherwise mediocre Variants 1 and 4. This is likely a manifestation of the efficiency of the underlying sequential GotoBLAS `dgemm` in performing matrix-multiply on operands with certain shapes. As mentioned previously, the `dgemm` used tends to perform worse on the row-panel matrix multiply found in the GEMM subproblems of Variants 1 and 4.

Benefits of 2D partitioning for small problems. Figure 16 organizes the data for Variant 2 present in Figure 15 in order to better contrast the four methods of task partitioning for small problems. In addition to showing absolute performance for each of the four task partitionings, the figure includes a graph

showing speedup relative to the proportional one-dimensional partitioning given by $b = (-24, -1)$. The x-axis range and data point increments have been decreased in order to show more detail. The figure's right-hand graph reveals that a two-dimensional partitioning with proportional blocking ($b = (-12, -2)$) outperforms a similar one-dimensional partitioning ($b = (-24, -1)$) for small problems.

More on constant blocking/2D partitioning. Finally, Figure 17 shows the effect of moving from a one-dimensional to a two-dimensional partitioning, for both constant and proportional blocking. The following observations may be made:

—In the case of moving from $b = (200, -1)$ to $b = (200, 200)$, we see that the performance for the two-dimensional partitioning rises more sharply for smaller problems but levels off lower than that of the 1D partitioning. This most likely is due to the reduced `dgemm` efficiency that comes with casting most of the computations in terms of small 200×200 problems. By contrast, the 1D partitioning, while suffering from poor load-balance early on, maintains higher efficiency due to the variable-sized tasks creating GEMM operations where one dimension is, on average, still relatively large.

—Figure 17 also shows the effects of using a smaller constant blocksize. This is shown for blocksize pairs $b = (100, -1)$ and $b = (100, 100)$. Not surprisingly, both outperform their larger counterparts for certain small problems. As the problem size increases, the blocksize pairs with smaller blocksize values create tasks more quickly, allowing more parallelism for smaller problems. However, these smaller blockings achieve a lower peak performance due to reduced `dgemm` efficiency.

—For proportional blocking, we see once again that a two-dimensional partitioning is beneficial for small problems. Also included in these two graphs are data for $b = (-8, -3)$ and $b = (-6, -4)$. The performance of these partitionings roughly matches that of $b = (-24, -1)$ and $b = (-12, -2)$ for small problems but suffers slightly for large matrices. We suspect that this effect is not due to a loss of `dgemm` efficiency but rather suboptimal load-balancing. In our discussion of Figure 14, we pointed out that the partitioning given by $b = (-12, -2)$ was more desirable than other proportional 2D partitionings. In this case, neither $b = (-8, -3)$ nor $b = (-6, -4)$ load-balances as well across 12 threads due to the fact that fewer fixed-sized SYRK subproblem tasks (8 and 6, respectively) are created than threads used in the computation. Furthermore, each of these SYRK tasks is rather large, raising the potential for threads to become idle while some remaining portion of the SYRK computation is still in progress. Presumably, these factors conspire to prevent a favorable scheduling for most larger problems.

7. CONCLUSION AND FUTURE DIRECTIONS

In this article, we discussed the high-performance parallel implementation of the symmetric rank-k update operation, targeting SMP and future multi-core

architectures. This operation is representative of how many level-3 BLAS and LAPACK-like operations are implemented with the FLAME/C API. Several contributions were reported that improve ease of implementation as well as performance.

We demonstrated how task queues, a proposed feature for OpenMP 3.0, allow code that is devoid of indexing to be elegantly and effectively parallelized. We identified and overcame two shortcomings present in the current Intel implementation of OpenMP task queues. First, we implemented a more portable domain-specific workqueuing solution for FLAME/C that uses only conventional OpenMP constructs. Second, we demonstrated the benefits of scheduling tasks in descending order of cost in the context of the SYRK operation. In addition, we demonstrated the merit in proportional blocking and found that a two-dimensional data partitioning gives way to better performance for smaller problems.

Both OpenMP and FLAME workqueuing implementations allow algorithms to be coded and parallelized at a much higher level of abstraction and, in our experience, improves almost all stages of library development. The resulting parallelized code was shown to require only minor modifications to the corresponding sequential FLAME/C implementation. Very good speedup was reported on a medium sized SMP system.

We believe this work provides the architects of OpenMP workqueuing with preliminary evidence that more control over task scheduling would benefit end-user performance. Specifically, these findings suggest that the workqueuing mechanism should allow the queue to be filled and sorted according to task cost before execution takes place. By including an optional `cost` clause in the `task` directive specification, a programmer could provide the implementation with an estimate for the cost of each task. This information would allow threads to dequeue tasks from largest to smallest, thereby potentially improving load-balance when tasks naturally vary in cost.

Future Work. After inspecting the GotoBLAS implementation of matrix-matrix multiplication [Goto and van de Geijn 2008], we have concluded that the proposed parallelization based on local GEMM operations causes threads to duplicate internal copying and packing of data. By exposing low-level interfaces to these underlying operations, it should be possible to schedule data movements so that duplication and/or memory contention can be reduced, yielding still better performance. It should be feasible to incorporate these insights into the methodologies discussed in the present article.

Further Information

For additional information regarding the FLAME project, visit

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

ACKNOWLEDGMENTS

The OpenMP task queue construct was brought to our attention by Dr. Timothy Mattson, Intel. This was the key insight that has allowed us to avoid the reintroduction of indices.

Access to the 14 CPU Itanium2, 1.5 GHz, system on which the experiments were performed was provided by Dr. Gregorio Quintana-Ortí of Universidad Jaume I, Spain. Some experiments were prepared and tested on a 4 CPU Itanium2, 1.5 GHz, server, which was generously donated to our research efforts by Hewlett-Packard and is administered by UT-Austin's Texas Advanced Computing Center.

We thank Kazushige Goto and Dr. Kent Milfeld, both with the Texas Advanced Computing Center, for their valuable feedback throughout our research. We also thank Dr. Andrew Chapman and Thuan Cao, both with NEC Solutions America, Inc., for their technical advice. We would like to acknowledge input from Dr. Enrique Quintana-Ortí on a draft of this article.

REFERENCES

Anderson, E., Bai, Z., Demmel, J., Dongarra, J. E., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A. E., Ostrouchov, S., and Sorensen, D. 1992. *LAPACK Users' Guide*. SIAM, Philadelphia.

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

Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S., and van de Geijn, R. A. 2005. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Softw. 31*, 1 (March), 1–26.

Bientinesi, P., Quintana-Ortí, E. S., and van de Geijn, R. A. 2005. Representing linear algebra algorithms in code: The FLAME APIs. *ACM Trans. Math. Softw. 31*, 1 (March), 27–59.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16*, 1 (March), 1–17.

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

Dow, E. 2005. Take charge of processor affinity. IBM developerWorks. `http://www.ibm.com/developerworks/linux/library/l-affinity.html`.

Garey, M. R., Graham, R. L., and Ullman, J. D. 1973. An analysis of some packing algorithms. In *Combinatorial Algorithms*, R. Rustin, Ed. Algorithmics Press, New York, 39–47.

Goto, K. 2006. `http://www.cs.utexas.edu/users/kgoto`.

Goto, K. and van de Geijn, R. A. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw. 34*, 3.

Johnson, D. S. 1973. Approximation algorithms for combinatorial problems. In *Fifth Annual ACM Symposium on Theory of Computing*. ACM, New York, 38–49.

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

Low, T. M., Milfeld, K. F., van de Geijn, R. A., and Van Zee, F. G. 2004. Parallelizing FLAME code with OpenMP task queues. Department of Computer Sciences Tech. Rep. TR-04-05, The University of Texas at Austin (December).

Low, T. M., van de Geijn, R. A., and Van Zee, F. G. 2005. 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)*. ACM Press, New York, NY, 153–163.

OpenMP Architecture Review Board. 2006. `http://www.openmp.org/`.

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

Shah, S., Haab, G., Peterson, P., and Throop, J. 1999. Flexible control structures for parallelism in OpenMP. In P2, 6, 7 *European Workshop on OpenMP* (*EWOMP*).

SU, E., TIAN, X., GIRKAR, M., HAAB, G., SHAH, S., AND PETERSON, P. 2002. Compiler support of the workqueuing execution model for Intel SMP architectures. In *European Workshop on OpenMP* (*EWOMP*).

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

WEISSTEIN, E. W. 2006. Bin-Packing Problem. From MathWorld—A Wolfram Web Resource. `http://mathworld.wolfram.com/Bin-PackingProblem.html`.

Received October 2006; revised April 2007; accepted April 2007

Programming Matrix Algorithms-by-Blocks for Thread-Level Parallelism

GREGORIO QUINTANA-ORTí and ENRIQUE S. QUINTANA-ORTí
Universidad Jaume I
and
ROBERT A. VAN DE GEIJN, FIELD G. VAN ZEE, and ERNIE CHAN
The University of Texas at Austin

With the emergence of thread-level parallelism as the primary means for continued performance improvement, the programmability issue has reemerged as an obstacle to the use of architectural advances. We argue that evolving legacy libraries for dense and banded linear algebra is not a viable solution due to constraints imposed by early design decisions. We propose a philosophy of abstraction and separation of concerns that provides a promising solution in this problem domain. The first abstraction, FLASH, allows algorithms to express computation with matrices consisting of contiguous blocks, facilitating algorithms-by-blocks. Operand descriptions are registered for a particular operation a priori by the library implementor. A runtime system, SuperMatrix, uses this information to identify data dependencies between suboperations, allowing them to be scheduled to threads out-of-order and executed in parallel. But not all classical algorithms in linear algebra lend themselves to conversion to algorithms-by-blocks. We show how our recently proposed LU factorization with incremental pivoting and a closely related algorithm-by-blocks for the QR factorization, both originally designed for out-of-core computation, overcome this difficulty. Anecdotal evidence regarding the development of routines with a core functionality demonstrates how the methodology supports high productivity while experimental results suggest that high performance is abundantly achievable.

This research was partially sponsored by NSF grants CCF-0540926 and CCF-072714. Additional support came from the J. Tinsley Oden Faculty Fellowship Research Program of the Institute for Computational Engineering and Sciences (ICES) at UT-Austin. Gregorio Quintana-Ortí and Enrique S. Quintana-Ortí were supported by the CICYT project TIN2005-09037-C02-02 and FEDER, and projects P1B-2007-19 and P1B-2007-32 of the Fundación Caixa-Castellón/Bancaixa and UJI. 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 (NSF).

Authors' addresses: G. Quintana-Ortí and E. S. Quintana-Ortí, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume 1, Campus Riu Sec, 12.071, Castellón, Spain; email: {gquintan,quintana}@icc.uji.es; R. A. van de Geijn, F. G. Van Zee, and E. Chan, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: {rvdg,field,echan}@ cs.utexas.edu.

DOI 10.1145/1527286.1527288 http://doi.acm.org/10.1145/1527286.1527288

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

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Linear algebra, libraries, high-performance, multithreaded architectures

ACM Reference Format:

Quintana-Ortí, G., Quintana-Ortí, E. S., van de Geijn, R. A., Van Zee, F. G., and Chan, E. 2009. Programming matrix algorithms-by-blocks for thread-level parallelism. ACM Trans. Math. Softw. 36, 3, Article 14 (July 2009), 26 pages.
DOI = 10.1145/1527286.1527288 http://doi.acm.org/10.1145/ 1527286.1527288

1. INTRODUCTION

For the first time in history, computer architectures are approaching physical and technological barriers which make increasing the speed of a single core exceedingly difficult and economically infeasible. As a result, hardware architects have begun to design microprocessors with multiple processing cores that operate independently and share the same address space. It appears that the advent of these multicore architectures will finally force a radical change in how applications are programmed. Specifically, developers must consider how to direct the collaboration of many concurrent threads of execution to solve a single problem. In this article, we present what we believe is a promising solution to what is widely perceived to be a very difficult problem, targeting the domain of dense and banded linear algebra libraries. The approach views submatrices (blocks) as units of data, algorithms as operating on these blocks (algorithms-by-blocks), and schedules the operations on blocks using out-of-order techniques similar to how superscalar processors schedule instructions and resolve dependencies on individual data.

Experience gained from parallel computing architectures with complex hierarchical memories has shown that an intricate and interdependent set of requirements must be met in order to achieve the level of performance that scientific applications demand of linear algebra libraries. These requirements include data locality, load balance, and careful attention to the critical path of computation, to name a few. While it may be possible to satisfy this set of constraints when implementing a single operation on a single architecture, addressing them for an entire library of commonly used operations requires one to face the additional challenge of programmability.

We propose abandoning essentially all programming conventions that were adopted when widely used libraries like LINPACK [Dongarra et al. 1979], LAPACK [Anderson et al. 1999], and ScaLAPACK [Choi et al. 1992], denoted by LIN(Sca)LAPACK hereafter, were designed in the last quarter of the 20th century.[1] To us, nothing associated with programming these libraries is sacred: not the algorithms, not the notation used to express algorithms, not the data structures used to store matrices, not the APIs used to code the algorithms,

[1]We do not mean to deminish the contributions of the LINPACK and LAPACK projects. We merely suggest that the programming styles used by those packages, while cutting-edge at the time they were developed, need to be reconsidered. We fully recognize that these projects made tremendous contributions to the field of numerical linear algebra beyond the packages that they delivered.

and not the runtime system that executes the implementations (if one ever existed). Instead we build on the notation, APIs, and tools developed as part of the FLAME project [van de Geijn and Quintana-Ortí 2008], which provide modern object-oriented abstractions to increase developer productivity and user friendliness alike. Perhaps the most influential of these abstractions is one that provides advanced shared-memory parallelism by borrowing out-of-order scheduling techniques from sequential superscalar architectures.

From the outset, we have been focused on developing a prototype library that encompasses much of the functionality of the core of LAPACK, including LU, QR and Cholesky factorizations, and related solvers. The LU factorization with pivoting and Householder QR factorizations are key since algorithms-by-blocks for these operations require new algorithms. This is due to the fact that when traditional partial pivoting or traditional Householder transformations are employed, entire columns have to be examined and/or updated, which becomes a synchronization point in the computation [Quintana-Ortí et al. 2008a, 2008b]. Fortunately, we had already developed high-performance algorithms for out-of-core computation that were algorithm-by-blocks, except that the block size targeted the movement of data from disk to main memory rather than thread-level parallelism [Gunter and van de Geijn 2005; Joffrain et al. 2004]. Thus, we knew from the start that algorithms-by-blocks were achievable for all of these important operations.

We focus on the programmability issue that developers face given that parallelism will be required in order to exploit the performance potential of multicore and many-core systems. We point to the fact that the parallelization effort described in this article was conceived in early September 2006. Remarkably, all results presented in this article were already achieved by September 2007. The key has been a clear separation of concerns between the code that implements the algorithms and the runtime system that schedules tasks.

We are not alone in recognizing the utility of a runtime system that dynamically schedules subproblems for parallel execution. The PLASMA project [Buttari et al. 2007, 2008], independent of our efforts but with LU and QR factorization algorithms that are similarly based on our earlier work on out-of-core algorithms, has developed a similar mechanism in the context of the QR factorization. Like the SuperMatrix system described here and in Chan et al. [2007a, 2007b, 2008], the PLASMA system enqueues operation subproblems as "tasks" on a queue, builds a directed acyclic graph (DAG) to encode dependencies, and then executes the subproblems as task dependencies become satisfied [Buttari et al. 2008]. However, as part of Buttari et al. [2008] they did not provide any source code—neither example code implementing the runtime-aware algorithm nor code implementing the runtime system itself. The same authors expanded on their work in Buttari et al. [2007] to include remarks and results for the Cholesky factorization and LU factorization with incremental pivoting based on our out-of-core algorithm [Joffrain et al. 2004], which we encourage the reader to study. In contrast to the PLASMA project, we directly address the programmability issue.

The primary contribution of the present article lies with the more comprehensive description of how abstraction can be exploited to solve the

programmability problem that faces linear algebra library development with the advent of multicore architectures. In doing so, this article references a number of conference publications [Chan et al. 2007a, 2007b, 2008; Quintana-Ortí et al. 2008a, 2008b, 2008c] that provide evidence in support of claims that we make. As such, this is also a survey article.

The article can be viewed as consisting of two parts. The first part describes the methodology. Section 2 provides a motivating example in the form of the LU factorization (without pivoting) and makes the case that the LIN(Sca)LAPACK design philosophies are conducive neither to ease of programming nor efficient parallelism. Section 3 discusses algorithms-by-blocks and the challenges they present and provides an overview of the FLASH API, including an interface for filling matrices that uses storage-by-blocks. Section 4 gives an overview of the SuperMatrix runtime parallel execution mechanism in the context of the LU factorization. Upon finishing this first part, the reader will understand the general approach and, hopefully, the opportunities that it enables. The second part of the article argues that the methodology has the potential of solving the programmability problem in the domain of dense and banded linear algebra libraries. It does so primarily by expanding upon results from our other recent conference publications related to the FLAME project, thereby providing better perspective on the new techniques. Section 5 recounts the authors' work in parallelizing the level-3 basic linear algebra subprograms (BLAS) operations using SuperMatrix. Section 6 expands the discussion of operations to the LU factorization with partial and incremental pivoting. This section goes into quite a bit more detail because it is important to show how new algorithms can be formulated so that the methodology can be exploited. Section 7 briefly summarizes how these results extend to advanced linear algebra operations. Performance results are reported in Section 8. Finally, Section 9 contains concluding remarks including a discussion of possible future extensions.

2. A MOTIVATING EXAMPLE: THE LU FACTORIZATION WITHOUT PIVOTING

The LU factorization (without pivoting) of an $n \times n$ matrix A is given by $A = LU$, where L is $n \times n$ unit lower triangular and U is $n \times n$ upper triangular. In traditional algorithms for this factorization, the triangular factors overwrite A, with the strictly lower triangular part of L stored on the subdiagonal elements of A and the upper triangular part of U stored on those elements of A on and above the diagonal. We denote this as $A := \{L\backslash U\}$.

2.1 A Typical Algorithm

In Figure 1 we give unblocked and blocked algorithms, in FLAME notation [Gunnels et al. 2001], for overwriting a matrix A with the triangular factors L and U. The unblocked algorithm on the left involves vector-vector and matrix-vector operations, which perform $O(1)$ floating-point arithmetic operations (flops) for every memory operation (memop). This ratio renders low performance on current cache-based processors as memops are considerably slower than flops on these architectures. The blocked algorithm on the right of that figure is likely to attain high performance since most computation is cast in

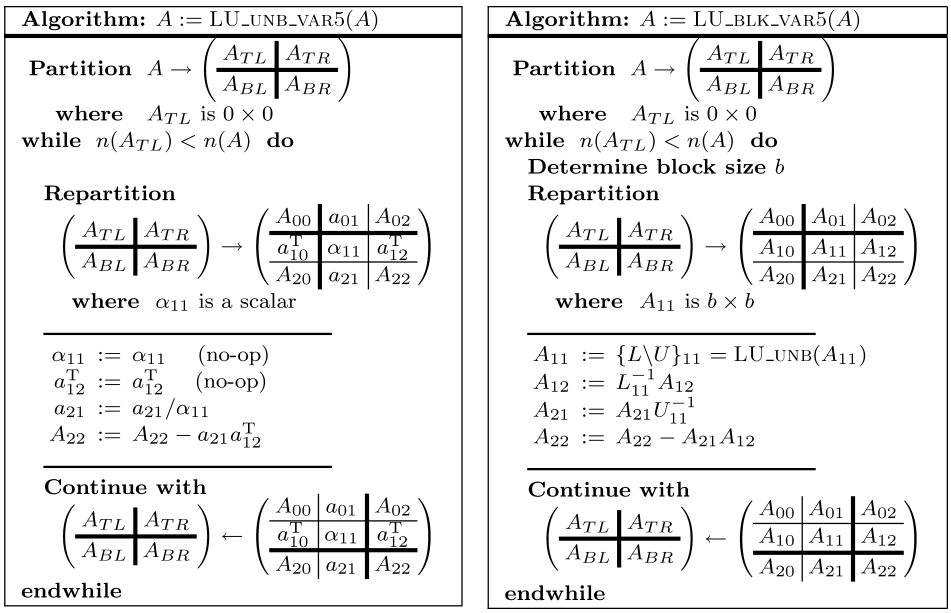


Fig. 1. Unblocked and blocked algorithms (left and right, respectively) for computing the LU factorization (without pivoting). Here, $n(B)$ stands for the number of columns of B.

terms of the *matrix-matrix product* (GEMM) $A_{22} := A_{22} - A_{21}A_{12}$, which performs $O(b)$ flops for every memop. In Gunnels et al. [2001], five algorithmic variants for computing the LU factorization were identified. The VAR5 that was part of the algorithm name indicated the given algorithm corresponded to Variant 5 in that article.

Using the FLAME/C API [Bientinesi et al. 2005], an equivalent blocked algorithm can be represented in code as presented in Figure 2 (left). Comparing and contrasting Figures 1 and 2 (left) shows how the FLAME notation, which departs from the commonly encountered loop-based algorithms, translates more naturally into code when an appropriate API is defined for the target programming language. And thus we abandon conventional algorithm notation and the LIN(Sca)LAPACK style of programming.

2.2 The Trouble with Evolving Legacy Code to Multithreaded Architectures

A commonly employed approach to parallelizing dense linear algebra operations on multithreaded architectures has been to push parallelism into multithreaded versions of the BLAS [Dongarra et al. 1988, 1990; Lawson et al. 1979]. The rationale behind this is to make minimal changes to existing codes.

In the case of the LU factorization, this means parallelism is attained only within the two TRSM and the GEMM operations:

$$\begin{aligned} A_{12} &:= L_{11}^{-1}A_{12}, \\ A_{21} &:= A_{21}U_{11}^{-1}, \\ A_{22} &:= A_{22} - A_{21}A_{12}. \end{aligned}$$

```
FLA_Error FLA_LU_blk_var5( FLA_Obj A, int nb_alg )
{
  FLA_Obj ATL, ATR,      A00, A01, A02,
          ABL, ABR,      A10, A11, A12,
                         A20, A21, A22;
  int b;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_width( ATL ) < FLA_Obj_width( A ) ) {
    b = min( FLA_Obj_length(ABR), nb_alg );
    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                            &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
       b, b, FLA_BR );
    /*-----------------------------------------------*/
    FLA_LU_unb_var5( A11 );
    FLA_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR,
              FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
              FLA_ONE, A11,
                       A12 );
    FLA_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR,
              FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
              FLA_ONE, A11,
                       A21 );
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A21,
                             A12,
              FLA_ONE,       A22 );
    /*-----------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                               A10, A11, /**/ A12,
    /* *************** */   /* ****************** */
       &ABL, /**/ &ABR,        A20, A21, /**/ A22,
       FLA_TL );
  }
  return FLA_SUCCESS;
}
```

```
FLA_Error FLASH_LU_by_blocks_var5( FLA_Obj A )
{
  FLA_Obj ATL, ATR,      A00, A01, A02,
          ABL, ABR,      A10, A11, A12,
                         A20, A21, A22;


  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_width( ATL ) < FLA_Obj_width( A ) ) {

    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                            &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
       1, 1, FLA_BR );
    /*-----------------------------------------------*/
    FLA_LU_unb_var5( FLASH_MATRIX_AT( A11 ) );
    FLASH_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR,
                FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
                FLA_ONE, A11,
                         A12 );
    FLASH_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR,
                FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
                FLA_ONE, A11,
                         A21 );
    FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
                FLA_MINUS_ONE, A21,
                               A12,
                FLA_ONE,       A22 );
    /*-----------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                               A10, A11, /**/ A12,
    /* *************** */   /* ****************** */
       &ABL, /**/ &ABR,        A20, A21, /**/ A22,
       FLA_TL );
  }
  return FLA_SUCCESS;
}
```

Fig. 2. Left: FLAME/C implementation of the blocked algorithm in Figure 1 (right). Right: FLASH implementation of the algorithm-by-blocks, which is described in Section 3.3.

While we will see that this works well when the matrix is large and there are relatively few processors, a bottleneck forms when the ratio of the matrix dimension to the number of processors is low. In particular, the block size (variable b in Figures 1 (right) and 2 (left), respectively) must be relatively large (in practice, in the 128–256 range) so that the GEMM subproblems, which form the bulk of the LU computation, deliver high performance [Goto and van de Geijn 2008]. As a result, the LU factorization of A_{11}, typically computed by only a single processor, leaves other threads idle and therefore hinders parallel efficiency. Thus, this approach to extracting parallelism is inherently limited.

One technique that attempts to overcome such a bottleneck is to "compute ahead." Consider the illustration in Figure 3 of the partitionings of A at the beginning of the first two iterations of the blocked algorithm for the LU factorization. In this technique, the update of A_{22} during the first iteration is broken down into the update of the part of A_{22} that will become A_{11} in the next iteration (see Figure 3), followed by the update of the rest of A_{22}. This then allows the factorization of the next A_{11} to be scheduled before the update of the remaining parts of the current A_{22}, thus overcoming the bottleneck. Extensions of this idea compute ahead several iterations in a similar manner.

The problem with this idea is that it greatly complicates the code that implements the algorithm if coded in a traditional style [Addison et al. 2003; Kurzak

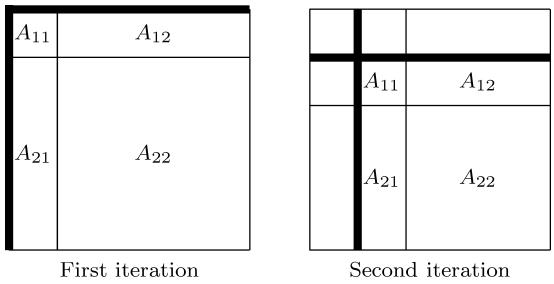


Fig. 3. First two iterations of the blocked algorithm in Figure 1 (right).

and Dongarra 2006; Strazdins 2001]. While feasible for a single, relatively simple algorithm like the LU factorization without pivoting or the Cholesky factorization, reimplementing a linear algebra library like LAPACK would become a daunting task if this strategy were employed.

3. ALGORITHMS-BY-BLOCKS

Fred Gustavson (IBM) has long advocated an alternative to the blocked algorithms in LAPACK [Agarwal and Gustavson 1989; Elmroth et al. 2004; Gustavson et al. 2007]. The solution, *algorithms-by-blocks*, proposes algorithms that view matrices as collections of submatrices and express their computation in terms of these submatrix blocks.

3.1 Basic Idea

The idea is simple. When moving from algorithms that cast most computation in terms of matrix-vector operations to algorithms that mainly operate in terms of matrix-matrix computations, rather than improving performance by aggregating the computation into matrix-matrix computations, the developer should raise the granularity of the data by replacing each element in the matrix by a submatrix (block). Algorithms are then written as before, except with scalar operations replaced by operations on the blocks.

For example, consider the LU factorization of the partitioned matrix:

$$A \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^{\mathrm{T}} & \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{ccc|c|ccc} \bar{\alpha}_{00} & \cdots & \bar{\alpha}_{0k} & \cdots & \bar{\alpha}_{0,n-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \hline \bar{\alpha}_{k0} & \cdots & \bar{\alpha}_{kk} & \cdots & \bar{\alpha}_{k,n-1} \\ \hline \vdots & \ddots & \vdots & \ddots & \vdots \\ \bar{\alpha}_{n-1,0} & \cdots & \bar{\alpha}_{n-1,k} & \cdots & \bar{\alpha}_{n-1,n-1} \end{array}\right),$$

where α_{11} and $\bar{\alpha}_{ij}$, $0 \leq i, j < n$, are all scalars. The unblocked algorithm in Figure 1 (left) can be turned into an algorithm-by-blocks by recognizing that, if

each element in the matrix is itself a matrix, as in

$$A \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{ccc|c|ccc} \bar{A}_{00} & \cdots & \bar{A}_{0K} & \cdots & \bar{A}_{0,N-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \hline \bar{A}_{K0} & \cdots & \bar{A}_{KK} & \cdots & \bar{A}_{K,N-1} \\ \hline \vdots & \ddots & \vdots & \ddots & \vdots \\ \bar{A}_{N-1,0} & \cdots & \bar{A}_{N-1,K} & \cdots & \bar{A}_{N-1,N-1} \end{array}\right),$$

where A_{11} and $\bar{A}_{ij}$, $0 \leq i, j < N$, are all $b \times b$ blocks, then the following occurs:

(1) $\alpha_{11} := \alpha_{11}$(no-op) becomes the LU factorization of the matrix element A_{11}:

$$A_{11} := \{L \backslash U\}_{11} = \{\bar{L} \backslash \bar{U}\}_{KK}.$$

(2) a_{21} becomes the column vector of blocks A_{21} so that $a_{21} := a_{21}/\alpha_{11}$ is replaced by a triangular solve with multiple right-hand sides with the updated upper triangular matrix in A_{11} and each of the blocks in A_{21}:

$$A_{21} := A_{21} U_{11}^{-1} = \begin{pmatrix} \bar{A}_{K+1,K} \\ \vdots \\ \bar{A}_{N-1,K} \end{pmatrix} \bar{U}_{KK}^{-1} = \begin{pmatrix} \bar{A}_{K+1,K} \bar{U}_{KK}^{-1} \\ \vdots \\ \bar{A}_{N-1,K} \bar{U}_{KK}^{-1} \end{pmatrix}.$$

(3) a_{12}^{T} becomes a row vector of blocks A_{12} so that $a_{12}^{\mathrm{T}} := a_{12}^{\mathrm{T}}$(no-op) becomes a triangular solve with multiple right-hand sides with the updated lower triangular matrix in A_{11} and each of the blocks in A_{12}:

$$\begin{aligned} A_{12} := L_{11}^{-1} A_{12} &= \bar{L}_{KK}^{-1} \left(\bar{A}_{K,K+1} \ \cdots \ \bar{A}_{K,N-1} \right) \\ &= \left(\bar{L}_{KK}^{-1} \bar{A}_{K,K+1} \ \cdots \ \bar{L}_{KK}^{-1} \bar{A}_{K,N-1} \right). \end{aligned}$$

(4) Each element in A_{22} describes a block that needs to be updated via a matrix-matrix product using blocks from the updated vectors of blocks A_{21} and A_{12}:

$$\begin{aligned} A_{22} &:= A_{22} - A_{21} A_{12} \\ &= \begin{pmatrix} \bar{A}_{K+1,K+1} & \cdots & \bar{A}_{K+1,N-1} \\ \vdots & \ddots & \vdots \\ \bar{A}_{N-1,K+1} & \cdots & \bar{A}_{N-1,N-1} \end{pmatrix} - \begin{pmatrix} \bar{A}_{K+1,K} \\ \vdots \\ \bar{A}_{N-1,K} \end{pmatrix} \left(\bar{A}_{K,K+1} \ \cdots \ \bar{A}_{K,N-1} \right) \\ &= \begin{pmatrix} \bar{A}_{K+1,K+1} - \bar{A}_{K+1,K} \bar{A}_{K,K+1} & \cdots & \bar{A}_{K+1,N-1} - \bar{A}_{K+1,K} \bar{A}_{K,N-1} \\ \vdots & \ddots & \vdots \\ \bar{A}_{N-1,K+1} - \bar{A}_{N-1,K} \bar{A}_{K,K+1} & \cdots & \bar{A}_{N-1,N-1} - \bar{A}_{N-1,K} \bar{A}_{K,N-1} \end{pmatrix}. \end{aligned}$$

Below we will show that the algorithm-by-blocks approach also facilitates the high-performance implementation and parallel execution of matrix operations on SMP and multicore architectures.

3.2 Obstacles

A major obstacle to algorithms-by-blocks lies with the complexity that is introduced into the code when matrices are manipulated and stored by blocks. A number of solutions have been proposed to solve this problem, ranging from

storing the matrix in arrays with four or more dimensions and explicitly exposing intricate indexing into the individual elements [Guo et al. 2008], to template programming using C++ [Valsalam and Skjellum 2002], and to compiler-based solutions [Wise et al. 2001]. None of these have yielded a consistent methodology that allows the development of high-performance libraries with functionality that rivals LAPACK or FLAME. The problem is programmability.

3.3 The FLASH API for Algorithms-by-Blocks

Several recent efforts [Elmroth et al. 2004; Herrero 2006; Low and van de Geijn 2004] have followed an approach different from those mentioned above. They viewed the matrix as a matrix of smaller matrices, just as it is conceptually described. Among these, the FLASH API [Low and van de Geijn 2004], which is an extension of the FLAME API, exploits the fact that FLAME encapsulates matrix information in objects by allowing elements of a matrix to themselves be descriptions of matrices. This approach supports (multilevel) hierarchical storage of matrices by submatrices (blocks). We note that, conceptually, these ideas are by no means new. The notion of storing matrices hierarchically goes back to the early 1970s [Skagestein 1972] and was rediscovered in the 1990s [Collins and Browne 1995].

Using the FLASH API, code for an algorithm-by-blocks for the LU factorization is given in Figure 2 (right). Note the similarity between that implementation and the blocked implementation for the LU factorization on the left of the same figure. That code maintains the traditional layering of subroutine calls that implement linear algebra operations, which illustrates that we are willing to preserve conventions from the BLAS/LIN(Sca)LAPACK efforts that continue to benefit programmability. However, it is worth pointing out that we actually advocate a deeper layering, one that allows the programmer to invoke routines that assume certain matrix shapes. This design yields potential performance benefits when the underlying BLAS implementation provides interfaces to low-level kernels [Goto and van de Geijn 2008; Marker et al. 2007]. Such an extended matrix multiplication interface can be seen in our use and implementation of `FLASH_Gebp_nn` in Figure 4, which assumes a matrix-matrix product where A is a block and B is a row panel.

It may seem that complexity is merely hidden in the routines `FLASH_Trsm` and `FLASH_Gemm`. The abbreviated implementations of these operations shown in Figure 4 demonstrate how the FLASH API is used in the implementation of those routines as well. The reader can see here that many of the details of the FLASH implementation have been buried within the FLASH-aware FLAME object definition. The fact that these algorithms operate on hierarchical matrices (which use storage-by-blocks) manifests itself only through the unit block size, the use of alternative `FLASH_` routines to further break subproblems into tasks with block operands, and an additional `FLASH_MATRIX_AT` macro to extract the appropriate submatrix when wrappers to external level-3 BLAS are invoked.

As a result, transforming blocked algorithms into algorithms-by-blocks and/or developing algorithms-by-blocks from scratch using the FLASH API is straightforward.

```
void FLASH_Trsm_llnu( FLA_Obj alpha, FLA_Obj L,
                                     FLA_Obj B )
/* Special case with mode parameters
     FLASH_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR,
                 FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
                 ...                                 )
   Assumption: L consists of one block and
               B consists of a row of blocks        */
{
  FLA_Obj BL, BR,      B0, B1, B2;



  FLA_Part_1x2( B,    &BL, &BR,    0, FLA_LEFT );


  while ( FLA_Obj_width( BL ) < FLA_Obj_width( B ) ) {
    FLA_Repart_1x2_to_1x3( BL, /**/ BR,   &B0, /**/ &B1, &B2,
                           1, FLA_RIGHT );


    /*-----------------------------------------------*/
    FLA_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR,
              FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
              alpha, FLASH_MATRIX_AT( L ),
                     FLASH_MATRIX_AT( B1 ) );
    /*-----------------------------------------------*/
    FLA_Cont_with_1x3_to_1x2( &BL, /**/ &BR,   B0, B1, /**/ B2,
                              FLA_LEFT );


  }
}
```

```
void FLASH_Trsm_runn( FLA_Obj alpha, FLA_Obj U,
                                     FLA_Obj B )
/* Special case with mode parameters
     FLASH_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR,
                 FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
                 ...                                 )
   Assumption: U consists of one block and
               B consists of a column of blocks     */
{
  FLA_Obj BT,      B0,
           BB,      B1,
                    B2;

  FLA_Part_2x1( B,    &BT,
                      &BB,    0, FLA_TOP );

  while ( FLA_Obj_length( BT ) < FLA_Obj_length( B ) ) {
    FLA_Repart_2x1_to_3x1( BT,       &B0,
                         /* ** */   /* ** */
                                    &B1,
                           BB,       &B2,    1, FLA_BOTTOM );
    /*-----------------------------------------------*/
    FLA_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR,
              FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
              alpha, FLASH_MATRIX_AT( U ),
                     FLASH_MATRIX_AT( B1 ) );
    /*-----------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &BT,        B0,
                                          B1,
                            /* ** */   /* ** */
                              &BB,        B2,    FLA_TOP );
  }
}
```

```
void FLASH_Gemm_nn( FLA_Obj alpha, FLA_Obj A,
                                   FLA_Obj B,
                    FLA_Obj beta,  FLA_Obj C )
/* Special case with mode parameters
     FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
                 ...                                 )
   Assumption: A is a column of blocks (column panel) and
               B is a row of blocks (row panel)     */
{
  FLA_Obj AT,      A0,      CT,      C0,
          AB,      A1,      CB,      C1,
                   A2,               C2;

  FLA_Part_2x1( A,    &AT,
                      &AB,    0, FLA_LEFT );
  FLA_Part_2x1( C,    &CT,
                      &CB,    0, FLA_TOP );

  while ( FLA_Obj_length( AL ) < FLA_Obj_length( A ) ) {
    FLA_Repart_2x1_to_3x1( AT,       &A0,
                         /* ** */   /* ** */
                                    &A1,
                           AB,       &A2,    1, FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( CT,       &C0,
                         /* ** */   /* ** */
                                    &C1,
                           CB,       &C2,    1, FLA_BOTTOM );
    /*-----------------------------------------------*/
    FLASH_Gebp_nn( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
                   alpha, A1,
                          B,
                   beta,  C1 );
    /*-----------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &AT,        A0,
                                          A1,
                            /* ** */   /* ** */
                              &AB,        A2,    FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &CT,        C0,
                                          C1,
                            /* ** */   /* ** */
                              &CB,        C2,    FLA_TOP );
  }
}
```

```
void FLASH_Gebp_nn( FLA_Obj alpha, FLA_Obj A,
                                   FLA_Obj B,
                    FLA_Obj beta,  FLA_Obj C )
/* Special case with mode parameters
     FLASH_Gebp( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
                 ...                                 )
   Assumption: A is a block and
               B, C are rows of blocks (row panels) */
{
  FLA_Obj BL, BR,      B0, B1, B2,
          CL, CR,      C0, C1, C2;


  FLA_Part_1x2( B,    &BL, &BR,    0, FLA_LEFT );

  FLA_Part_1x2( C,    &CL, &CR,    0, FLA_LEFT );


  while ( FLA_Obj_width( BL ) < FLA_Obj_width( B ) ) {
    FLA_Repart_1x2_to_1x3( BL, /**/ BR,   &B0, /**/ &B1, &B2,
                           1, FLA_RIGHT );


    FLA_Repart_1x2_to_1x3( CL, /**/ CR,   &C0, /**/ &C1, &C2,
                           1, FLA_RIGHT );


    /*-----------------------------------------------*/
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              alpha, FLASH_MATRIX_AT( A ),
                     FLASH_MATRIX_AT( B1 ),
              beta,  FLASH_MATRIX_AT( C1 ) );
    /*-----------------------------------------------*/
    FLA_Cont_with_1x3_to_1x2( &BL, /**/ &BR,   B0, B1, /**/ B2,
                              FLA_LEFT );


    FLA_Cont_with_1x3_to_1x2( &CL, /**/ &CR,   C0, C1, /**/ C2,
                              FLA_LEFT );


  }
}
```

Fig. 4. Code for the routines `FLASH_Trsm` and `FLASH_Gemm` that are needed to complete the LU factorization algorithm-by-blocks in Figure 2 (right). Top: FLASH implementations of triangular system solve with multiple right-hand sides. Bottom: FLASH implementations of matrix-matrix product.

3.4 Filling the Matrix

A nontrivial matter that has prevented acceptance of alternative data structures for storing matrices has been the interface to the application. Historically, the LIN(Sca)LAPACK approach has granted application direct access to the data. This requires the application programmer to understand how data is stored, which greatly increases the programming burden on the user [Edwards and van de Geijn 2006] particularly when matrices are distributed across processors, as is true for ScaLAPACK, or stored by blocks as discussed in the current article.

Our approach currently supports three alternative solutions.

—*Referencing conventional arrays*. Recall that the FLASH API allows matrix elements to contain submatrices. Leaf objects in this hierarchy encapsulate the actual numerical matrix data. Given a matrix stored in conventional column-major order, a FLASH matrix object can be constructed such that the leaf matrices simply refer to submatrices of the user-supplied matrix. Notice that this means the user can access the elements of the matrix as one would when interfacing with a conventional library like LAPACK. The main disadvantage is that the leaf matrices are not stored contiguously.

—*Contributions to a matrix object*. We will see that there is a distinct performance benefit to storing leaf matrices contiguously. Also, applications often naturally generate matrices by computing submatrices which are contributed to a larger overall matrix, possibly by adding to a partial result [Edwards and van de Geijn 2006].

For this scenario we provide routines for contribution to a FLASH matrix object. For example, FLASH provides a function whose signature is given by

```
void FLASH_Axpy_submatrix_to_global( FLA_Obj alpha,
                                     int m, int n,
                                     void* B, int ldb,
                                     int i, int j, FLA_Obj H);
```

This call accepts an `m` × `n` matrix B, stored at address `B` with leading dimension `ldb`, scales it by scalar α, and adds the result to the submatrix of H that has as its top-left element the (`i`, `j`) element of H. Note that matrix descriptor `H` refers to a hierarchically stored matrix object while B is a matrix created and stored in conventional column-major order. In (MATLAB) Mscript notation, this operation is given by

```
H( i:i+m-1, j:j+n-1 ) = alpha * B + H( i:i+m-1, j:j+n-1 );
```

A complementary routine allows submatrices to be extracted. Notice that given such an interface the user does not need to know how matrix H is actually stored.

This approach has been highly successful for interface applications to our Parallel Linear Algebra Package (PLAPACK) library for distributed-memory architectures [Edwards and van de Geijn 2006; van de Geijn 1997] where filling distributed matrices poses a similar challenge. Analogous interfaces are also used by the Global Array abstraction [Nieplocha et al. 1996] and

PETSc [Balay et al. 2004]. We strongly recommend this departure from the LIN/(Sca)LAPACK interface to applications.

—*Converting whole matrices*. It is also possible to allow the user to construct whole matrices in column-major order, which then may be used to build hierarchical matrices that contain the equivalent data. The submission process described above can be used for this conversion.

4. SUPERMATRIX OUT-OF-ORDER SCHEDULING

In this section we discuss how techniques used in superscalar processors can be adopted to systematically expose parallelism in algorithms-by-blocks without obfuscating the coded algorithms with further complexity.

4.1 SuperMatrix Dynamic Scheduling and Out-of-Order Execution

In order to illustrate the scheduling mechanism during this subsection, we consider the matrix of 3×3 blocks

$$A \rightarrow \begin{pmatrix} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} \end{pmatrix},$$

where all blocks are $b \times b$. First, the code in Figure 2 (right) is linked to the SuperMatrix runtime library and executed sequentially. As suboperations are encountered, the information associated with each suboperation is encapsulated and placed onto an internal queue. Once all operations are enqueued, the initial *analyzer stage* of execution is complete. Figure 5 provides a human-readable list corresponding to the full queue generated for a 3×3 matrix of blocks.

For example, during the first iteration of the code, the call

```
FLA_LU_unb_var5( FLASH_MATRIX_AT( A11 ) );
```

inserts the LU factorization of block $\bar{A}_{00}$ onto the list. During the same iteration, suboperations encountered inside `FLASH_Trsm` or `FLASH_Syrk` also enqueue their corresponding task entries. The order of the operations in the list, together with the operands that are read (input operands) and written (output operands) in the operation, determine the dependencies among matrix operations. Thus, the second operation in the list, which has $\bar{A}_{00}$ as an input operand and $\bar{A}_{01}$ as both an input and an output operand, requires the first operation to be completed before it may begin. The list denotes available operands with a "$\surd$" symbol; these operands are not dependent upon the completion of any other operations. They also happen to represent the operands that are available at the beginning of the algorithm-by-blocks since the list captures the state of the queue before execution.

During the *scheduler/dispatcher stage*, operations that have all operands available are scheduled for execution. As computation progresses, dependencies are satisfied and new operands become available, allowing more operations to be dequeued and executed (see Figure 6). The overhead of this runtime mechanism is amortized over a large amount of computation, and therefore its overall cost is minor.

Operation	Result	Operands		
		In		In/out
1. FLA_LU_unb_var5($\bar{A}_{00}$)	$\bar{A}_{00} := \{\bar{L}_{00} \backslash \bar{U}_{00}\} = LU(\bar{A}_{00})$			$\bar{A}_{00}\surd$
2. FLA_Trsm($\ldots, \bar{A}_{00}, \bar{A}_{01}$)	$\bar{A}_{01} := \bar{L}_{00}^{-1}\bar{A}_{01}$	$\bar{A}_{00}$		$\bar{A}_{01}\surd$
3. FLA_Trsm($\ldots, \bar{A}_{00}, \bar{A}_{02}$)	$\bar{A}_{02} := \bar{L}_{00}^{-1}\bar{A}_{02}$	$\bar{A}_{00}$		$\bar{A}_{02}\surd$
4. FLA_Trsm($\ldots, \bar{A}_{00}, \bar{A}_{10}$)	$\bar{A}_{10} := \bar{A}_{10}\bar{U}_{00}^{-1}$	$\bar{A}_{00}$		$\bar{A}_{10}\surd$
5. FLA_Trsm($\ldots, \bar{A}_{00}, \bar{A}_{20}$)	$\bar{A}_{20} := \bar{A}_{20}\bar{U}_{00}^{-1}$	$\bar{A}_{00}$		$\bar{A}_{20}\surd$
6. FLA_Gemm($\ldots, \bar{A}_{10}, \bar{A}_{01}, \ldots, \bar{A}_{11}$)	$\bar{A}_{11} := \bar{A}_{11} - \bar{A}_{10}\bar{A}_{01}$	$\bar{A}_{10}$	$\bar{A}_{01}$	$\bar{A}_{11}\surd$
7. FLA_Gemm($\ldots, \bar{A}_{10}, \bar{A}_{02}, \ldots, \bar{A}_{12}$)	$\bar{A}_{12} := \bar{A}_{12} - \bar{A}_{10}\bar{A}_{02}$	$\bar{A}_{10}$	$\bar{A}_{02}$	$\bar{A}_{12}\surd$
8. FLA_Gemm($\ldots, \bar{A}_{20}, \bar{A}_{01}, \ldots, \bar{A}_{21}$)	$\bar{A}_{21} := \bar{A}_{21} - \bar{A}_{20}\bar{A}_{01}$	$\bar{A}_{20}$	$\bar{A}_{01}$	$\bar{A}_{21}\surd$
9. FLA_Gemm($\ldots, \bar{A}_{20}, \bar{A}_{02}, \ldots, \bar{A}_{22}$)	$\bar{A}_{22} := \bar{A}_{22} - \bar{A}_{20}\bar{A}_{02}$	$\bar{A}_{20}$	$\bar{A}_{02}$	$\bar{A}_{22}\surd$
10. FLA_LU_unb_var5($\bar{A}_{11}$)	$\bar{A}_{11} := \{\bar{L}_{11} \backslash \bar{U}_{11}\} = LU(\bar{A}_{11})$			$\bar{A}_{11}$
11. FLA_Trsm($\ldots, \bar{A}_{11}, \bar{A}_{12}$)	$\bar{A}_{12} := \bar{L}_{11}^{-1}\bar{A}_{12}$	$\bar{A}_{11}$		$\bar{A}_{12}$
12. FLA_Trsm($\ldots, \bar{A}_{11}, \bar{A}_{21}$)	$\bar{A}_{21} := \bar{A}_{21}\bar{U}_{11}^{-1}$	$\bar{A}_{11}$		$\bar{A}_{21}$
13. FLA_Gemm($\ldots, \bar{A}_{21}, \bar{A}_{12}, \ldots, \bar{A}_{22}$)	$\bar{A}_{22} := \bar{A}_{22} - \bar{A}_{21}\bar{A}_{12}$	$\bar{A}_{21}$	$\bar{A}_{12}$	$\bar{A}_{22}$
14. FLA_LU_unb_var5($\bar{A}_{22}$)	$\bar{A}_{22} := \{\bar{L}_{22} \backslash \bar{U}_{22}\} = LU(\bar{A}_{22})$			$\bar{A}_{22}$

Fig. 5. Complete list of operations to be performed on blocks for the LU factorization (without pivoting) of a 3×3 matrix of blocks using algorithm-by-blocks. The "$\surd$"symbols denote those operands that are available immediately at the beginning of the algorithm (i.e., those operands that are not dependent upon other operations).

Thus, we combine two techniques from superscalar processors, dynamic scheduling and out-of-order execution, while hiding the management of data dependencies from both library developers and users. This approach is similar in philosophy to the inspector–executor paradigm for parallelization [Lu et al. 1997; von Hanxleden et al. 1992], but that work solves a very different problem. This approach also reflects a shift from control-level parallelism, specified strictly by the order in which operations appear in the code, to data-flow parallelism, which is restricted only by true data dependencies and availability of compute resources.

5. AN EXPERIMENT IN PROGRAMMABILITY: THE LEVEL-3 BLAS

In Chan et al. [2007a], we reported on the implementation of the level-3 BLAS using FLASH and SuperMatrix. In this section we briefly summarize the insights from that article with a primary focus on what it tells us about how the approach addresses the programmability issue.

When we commensed parallelizing the level-3 BLAS, we had a full set of sequential level-3 BLAS implemented using the FLAME API. It is important to realize that a "full set" entails all datatypes[2] and all unblocked and

[2]This includes single precision and double precision for real and complex operations.

	Original table			After 1st oper.			After 5th oper.			After 9th oper.		
Operation/Result	In		In/out	In		In/out	In		In/out	In		In/out
1. $LU(\bar{A}_{00})$			$\bar{A}_{00}\surd$									
2. $\text{TRISL}(\bar{A}_{00})^{-1}\bar{A}_{01}$	$\bar{A}_{00}$		$\bar{A}_{01}\surd$	$\bar{A}_{00}\surd$		$\bar{A}_{01}\surd$						
3. $\text{TRISL}(\bar{A}_{00})^{-1}\bar{A}_{02}$	$\bar{A}_{00}$		$\bar{A}_{02}\surd$	$\bar{A}_{00}\surd$		$\bar{A}_{02}\surd$						
4. $\bar{A}_{10}\text{TRIU}(\bar{A}_{00})^{-1}$	$\bar{A}_{00}$		$\bar{A}_{10}\surd$	$\bar{A}_{00}\surd$		$\bar{A}_{10}\surd$						
5. $\bar{A}_{20}\text{TRIU}(\bar{A}_{00})^{-1}$	$\bar{A}_{00}$		$\bar{A}_{20}\surd$	$\bar{A}_{00}\surd$		$\bar{A}_{20}\surd$						
6. $\bar{A}_{11}-\bar{A}_{10}\bar{A}_{01}$	$\bar{A}_{10}$	$\bar{A}_{01}$	$\bar{A}_{11}\surd$	$\bar{A}_{10}$	$\bar{A}_{01}$	$\bar{A}_{11}\surd$	$\bar{A}_{10}\surd$	$\bar{A}_{01}\surd$	$\bar{A}_{11}\surd$			
7. $\bar{A}_{12}-\bar{A}_{10}\bar{A}_{02}$	$\bar{A}_{10}$	$\bar{A}_{02}$	$\bar{A}_{12}\surd$	$\bar{A}_{10}$	$\bar{A}_{02}$	$\bar{A}_{12}\surd$	$\bar{A}_{10}\surd$	$\bar{A}_{02}\surd$	$\bar{A}_{12}\surd$			
8. $\bar{A}_{21}-\bar{A}_{20}\bar{A}_{01}$	$\bar{A}_{20}$	$\bar{A}_{01}$	$\bar{A}_{21}\surd$	$\bar{A}_{20}$	$\bar{A}_{01}$	$\bar{A}_{21}\surd$	$\bar{A}_{20}\surd$	$\bar{A}_{01}\surd$	$\bar{A}_{21}\surd$			
9. $\bar{A}_{22}-\bar{A}_{20}\bar{A}_{02}$	$\bar{A}_{20}$	$\bar{A}_{02}$	$\bar{A}_{22}\surd$	$\bar{A}_{20}$	$\bar{A}_{02}$	$\bar{A}_{22}\surd$	$\bar{A}_{20}\surd$	$\bar{A}_{02}\surd$	$\bar{A}_{22}\surd$			
10. $LU(\bar{A}_{11})$			$\bar{A}_{11}$			$\bar{A}_{11}$			$\bar{A}_{11}$			$\bar{A}_{11}\surd$
11. $\text{TRISL}(\bar{A}_{11})^{-1}\bar{A}_{12}$	$\bar{A}_{11}$		$\bar{A}_{12}$	$\bar{A}_{11}$		$\bar{A}_{12}$	$\bar{A}_{11}$		$\bar{A}_{12}$	$\bar{A}_{11}$		$\bar{A}_{12}\surd$
12. $\bar{A}_{21}\text{TRIU}(\bar{A}_{11})^{-1}$	$\bar{A}_{11}$		$\bar{A}_{21}$	$\bar{A}_{11}$		$\bar{A}_{21}$	$\bar{A}_{11}$		$\bar{A}_{21}$	$\bar{A}_{11}$		$\bar{A}_{21}\surd$
13. $\bar{A}_{22}-\bar{A}_{21}\bar{A}_{12}$	$\bar{A}_{21}$	$\bar{A}_{12}$	$\bar{A}_{22}$	$\bar{A}_{21}$	$\bar{A}_{12}$	$\bar{A}_{22}$	$\bar{A}_{21}$	$\bar{A}_{12}$	$\bar{A}_{22}$	$\bar{A}_{21}$	$\bar{A}_{12}$	$\bar{A}_{22}\surd$
14. $LU(\bar{A}_{22})$			$\bar{A}_{22}$			$\bar{A}_{22}$			$\bar{A}_{22}$			$\bar{A}_{22}$

Fig. 6. An illustration of the scheduling of operations for the LU factorization (without pivoting) of a 3×3 matrix of blocks using algorithm-by-blocks. Here, $\text{TRIU}(B)$ stands for the upper triangular part of B while $\text{TRISL}(B)$ denotes the matrix consisting of the lower triangular part of B with the diagonal entries replaced by ones.

blocked algorithm variants[3] for all operations that constitute the level-3 BLAS. We also had an implementation of the FLASH extension to FLAME and the SuperMatrix runtime system for scheduling the tasks used for the execution of a SuperMatrix-enabled algorithm. This implementation had previously been used only for the Cholesky factorization.

Two of the authors, Ernie Chan and Field Van Zee, spent a weekend implementing and testing the parallelization, yielding a full set of multithreaded level-3 BLAS using FLASH and SuperMatrix. This productivity attests to how effectively the methodology addresses the programmability issue. Impressive performance was reported in Chan et al. [2007b] despite the absence of dependencies in many of the reported operations, the lack of which reduces much of the SuperMatrix system to overhead.

6. LAPACK-LEVEL OPERATIONS: DENSE FACTORIZATIONS

The LAPACK library provides functionality one level above the level-3 BLAS. The subset with which we will primarily concern ourselves in this section includes the LU with pivoting, QR, and Cholesky factorizations.

6.1 An Algorithm-by-Blocks for the LU Factorization with Pivoting

It becomes immediately obvious that algorithms-by-blocks for the LU factorization with partial pivoting and the QR factorization based on Householder

[3]The FLAME methodology often yields half a dozen or more algorithms for each operation.

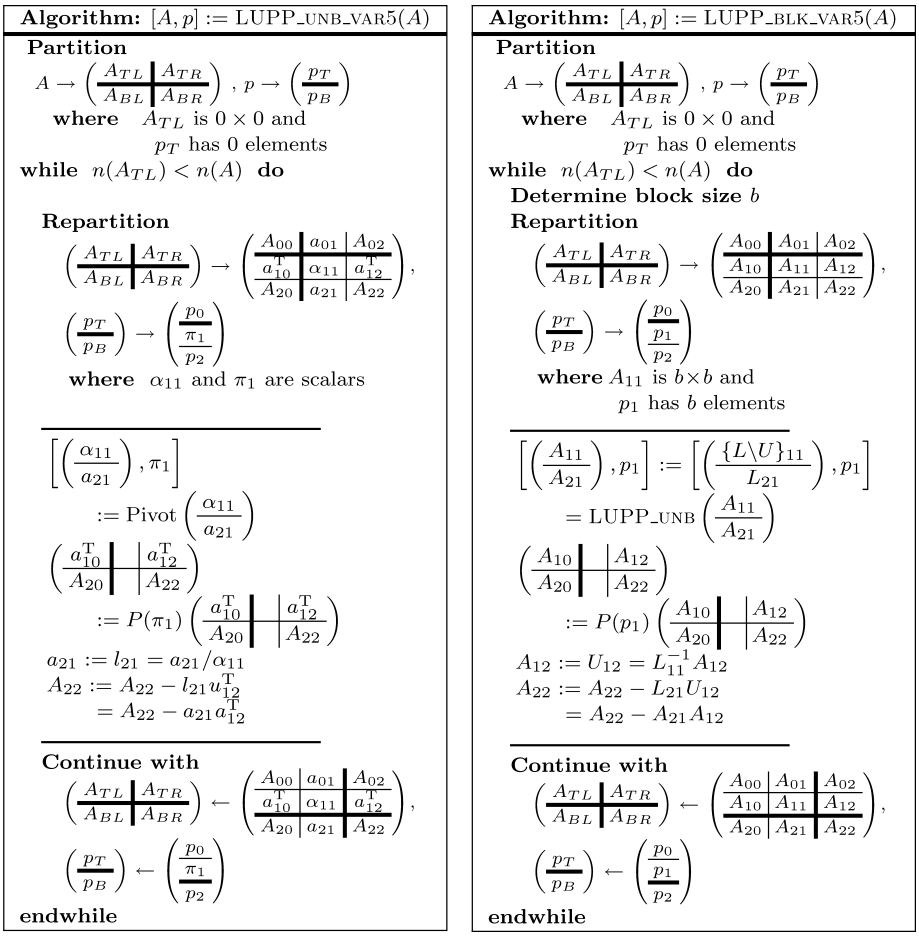


Fig. 7. Unblocked and blocked algorithms (left and right, respectively) for computing the LU factorization with partial pivoting. Pivot(v) refers to a function that returns the index of the entry of largest magnitude of a vector v and interchanges that element with the first entry of v. $P(\pi_1)$ and $P(p_1)$ denote permutation matrices formed from the rows interchanges registered in π_1 and p_1, respectively.

transformations require us to abandon the tried-and-trusted algorithms incorporated in LAPACK since pivoting information for the former and the computation of Householder transformations for the latter require access to columns that span multiple blocks. Unblocked and blocked algorithms for the LU factorization with partial pivoting, in FLAME notation, are given in Figure 7.

Thus, a second major obstacle to algorithms-by-blocks is that not all operations lend themselves nicely to this class of algorithms, with a clear example being the LU factorization when pivoting for stability enters the picture. We next describe our solution to this problem, inspired by out-of-core tiled algorithms for the QR and LU factorizations [Gunter and van de Geijn 2005; Joffrain et al. 2004; Quintana-Ortí and van de Geijn 2009].

Traditional algorithms for the LU factorization with partial pivoting exhibit the property that an updated column is required for a critical computation; in order to compute which row to pivot during the kth iteration, the kth column must have been updated with respect to all previous computation. This greatly restricts the order in which the computations can be performed. The problem is compounded by the fact that the column needed for computing which row to pivot, as well as the row to be pivoted, likely spans multiple blocks. This need for viewing and/or storing matrices by blocks was also observed for out-of-core dense linear algebra computations [Toledo 1999] and the implementation of dense linear algebra operations on distributed-memory architectures [Choi et al. 1992; van de Geijn 1997].

We will describe how partial pivoting can be modified to facilitate an algorithm-by-blocks. We do so by first reviewing results from Joffrain et al. [2004] and Quintana-Ortí and van de Geijn [2009] that show how an LU factorization can be updated while incorporating pivoting. Afterward, we generalize the insights to the desired algorithm-by-blocks.

6.1.1 *Updating an LU Factorization.* We briefly review how to compute the LU factorization of a matrix A of the form

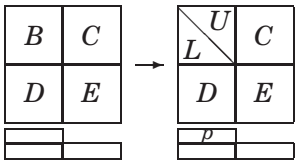
$$A = \left(\begin{array}{c|c} B & C \\ \hline D & E \end{array}\right) \tag{1}$$

in such a way that the LU factorization with partial pivoting of B can be reused if D, C, and E change. In our description, we assume that both B and E are square matrices.

The following procedure [Joffrain et al. 2004; Quintana-Ortí and van de Geijn 2009], consisting of five steps, computes an *LU factorization with incremental pivoting* of the matrix in (1):

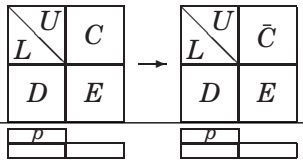
Step 1: Factor B. Compute the LU factorization with partial pivoting of B:

$$[B, p] := [\{L\backslash U\}, p] = \text{LUPP\_BLK}(B).$$



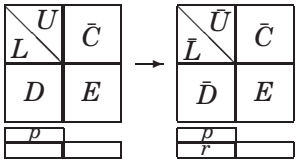
Step 2: Update C consistent with the factorization of B (using forward substitution):

$$\bar{C} := L^{-1}P(p)C = \text{TRSM\_LLNU}(L, P(p)C).$$



Step 3: Factor $\left(\frac{U}{D}\right)$. Compute the LU factorization with partial pivoting:

$$\left[\left(\frac{U}{D}\right), r\right] := \left[\left(\frac{\{\bar{L}\backslash\bar{U}\}}{\bar{D}}\right), r\right] = \text{LUPP\_SA\_BLK}\left(\left(\frac{U}{D}\right)\right).$$

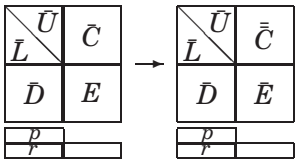


Here, $\bar{U}$ overwrites the upper triangular part of B (where U was stored before this operation). The lower triangular matrix $\bar{L}$ that results needs to be stored separately since both L, computed in Step 1 and used at Step 2, and $\bar{L}$ are needed during the forward substitution stage when solving a linear system.

Care must be taken in this step not to completely fill the zeroes below U, which would greatly increase the computational cost of the next step and the storage costs of both this and the next step. The procedure computes a "structure-aware" (SA) LU factorization with partial pivoting, employing a blocked algorithm that combines the LINPACK and LAPACK styles of pivoting. For details, see algorithm $\text{LU}_{\text{BLK}}^{\text{SA-LIN}}$ in Quintana-Ortí and van de Geijn [2009].

Step 4: Update $\left(\frac{\bar{C}}{E}\right)$ consistent with the factorization of $\left(\frac{U}{D}\right)$:

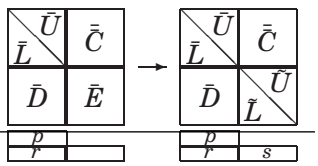
$$\left(\frac{\bar{C}}{\bar{E}}\right) := \left(\begin{array}{c|c} \bar{L} & 0 \\ \hline \check{L} & I \end{array}\right)^{-1} P(r)\left(\frac{\bar{C}}{E}\right) = \text{TRSM\_SA\_LLNU}\left(\left(\begin{array}{c|c} \bar{L} & 0 \\ \hline \check{L} & I \end{array}\right), P(r)\left(\frac{\bar{C}}{E}\right)\right).$$



Again, care must be taken in this step to exploit the zeroes below the diagonal of the upper triangular matrix produced in the previous step. This structure-aware procedure, though not equivalent to a clean triangular system solve (plus the application of the corresponding permutations), can be performed in terms of level-3 BLAS and presents essentially the same computational cost, modulo a lower-order term. For details, see algorithm $\text{FS}_{\text{BLK}}^{\text{SA-LIN}}$ in Quintana-Ortí and van de Geijn [2009].

Step 5: Factor E. Finally, compute the LU factorization with partial pivoting:

$$\left[\bar{\bar{E}}, s\right] := \left[\{\tilde{L}\backslash\tilde{U}\}, s\right] = \text{LUPP\_BLK}(\bar{E}).$$



for $k = 0 : N-1$
  $[A_{kk}, p_{kk}] := \text{LUPP\_BLK}(A_{kk})$
  for $j = k+1 : N-1$
    $A_{kj} := \text{TRSM\_LLNU}(A_{kk}, P(p_{kk})A_{kj})$
  endfor
  for $i = k+1 : N-1$
    $\left[\left(\frac{A_{kk}}{A_{ik}}\right), L_{ik}, p_{ik}\right] := \text{LUPP\_SA\_BLK}\left(\frac{A_{kk}}{A_{ik}}\right)$
    for $j = k+1 : N-1$
      $\left[\left(\frac{A_{kj}}{A_{ij}}\right)\right] := \text{TRSM\_SA\_LLNU}\left(\text{TRIL}\left(\left(\frac{L_{ik}\,|\,0}{A_{ik}\,|\,I}\right)\right),\ P(p_{ik})\left(\frac{A_{kj}}{A_{ij}}\right)\right)$
    endfor
  endfor
endfor

Fig. 8. Algorithm-by-blocks for the LU factorization with incremental pivoting. Here, TRIL(B) stands for the lower triangular part of B. The actual implementation is similar to those in Figures 2 (right) and 4, but for conciseness we present it as loops.

Overall, the five steps of the procedure apply Gauss transforms and permutations to reduce A to the upper triangular matrix

$$\left(\begin{array}{c|c} \bar{U} & \bar{C} \\ \hline 0 & \tilde{U} \end{array}\right).$$

6.1.2 *An Algorithm-by-Blocks.* The insights from the previous section naturally extend to an algorithm-by-blocks for the LU factorization with incremental pivoting [Joffrain et al. 2004; Quintana-Ortí et al. 2008a]. Consider the partitioning by *blocks*

$$A = \begin{pmatrix} A_{00} & A_{01} & \cdots & A_{0,N-1} \\ A_{10} & A_{11} & \cdots & A_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N-1,0} & A_{N-1,0} & \cdots & A_{N-1,N-1} \end{pmatrix}, \tag{2}$$

where, for simplicity, A_{ij}, $0 \leq i, j < N$, are considered to be of size $b \times b$. Then the algorithm in Figure 8 is a generalization of the algorithm described in Section 6.1.1.

While there is some flexibility in the order in which the loops are arranged, the SuperMatrix runtime system, described in Section 4, rearranges the operations, and therefore the exact order of the loops is not important.

6.1.3 *Stability.* Strictly speaking, the LU factorization with partial pivoting is not numerically stable; theory predicts that so-called element growth proportional to 2^n may occur. It is *practice* that taught us to rely on this method. In Quintana-Ortí and van de Geijn [2009], we discussed how the stability of incremental pivoting relates to that of partial pivoting and pairwise pivoting [Sorensen 1985]. In summary, incremental pivoting is a blocked variant of pairwise pivoting, being equivalent to partial pivoting when the block size equals the matrix dimension and to pairwise pivoting when the block size is 1. Element growth for partial pivoting is known to be bounded by 4^n. Therefore, element growth for incremental pivoting can be expected to be bounded by

a factor proportional to 2^n and 4^n, depending on the block size. The results in Quintana-Ortí and van de Geijn [2009] provide evidence in support of this observation. However, as was the case for partial pivoting, further practical experience will be needed to establish incremental pivoting as being a numerically stable method.

6.2 An Algorithm-by-Blocks for the QR Factorization

The QR factorization of an $m \times n$ matrix is given by $A = QR$, where Q is an $m \times m$ orthogonal matrix and R is an $m \times n$ upper triangular matrix. Although there exist several approaches to compute this factorization, the algorithm based on Householder reflectors [Golub and Loan 1996] is usually chosen when seeking high performance.

The QR factorization based on Householder reflectors and the LU factorization with partial pivoting share the property that an updated column is required for a critical computation; in the case of the QR factorization, the kth column must have been updated with respect to all previous computation before the Householder reflectors that annihilate subdiagonal entries in this column can be computed. This greatly restricts the order in which the computations can be performed.

An algorithm-by-blocks for the QR factorization can be obtained following the out-of-core algorithm in Gunter and van de Geijn [2005].

The algorithm-by-blocks for the QR factorization incurs a certain extra cost when compared with the traditional implementation of the QR factorization via Householder reflectors. This overhead is negligible for matrices of medium and large size. The use of orthogonal transformations ensures that the algorithm-by-blocks and the traditional QR factorization are numerically stable.

For details, see Quintana-Ortí et al. [2008b].

6.3 An Algorithm-by-Blocks for the Cholesky Factorization

Given a symmetric positive-definite (SPD) matrix A, its Cholesky factorization is given by $A = LL^T$ (or $A = U^T U$), where L is lower triangular (or U is upper triangular). The construction of an algorithm-by-blocks to obtain the Cholesky factorization is straightforward. The algorithm is illustrated in Figure 9 for an SPD matrix partitioned as in (2). On completion, the lower triangular part of A is overwritten by the Cholesky factor L while the strictly upper triangular part of the matrix is not modified.

This algorithm incurs the same flop count as the traditional implementation for the Cholesky factorization and the two exhibit the same numerical stability properties.

For details, see Chan et al. [2007a].

7. ADVANCED LAPACK-LEVEL OPERATIONS: BANDED FACTORIZATION AND INVERSION OF MATRICES

In this section, we offer a few comments on the development of algorithms-by-blocks for slightly more complex operations: factorization of a banded matrix and matrix inversion.

```
for k = 0 : N − 1
    [A_kk] := {L_kk \ A_kk} = CHOL_BLK(A_kk)
    for i = k + 1 : N − 1
        A_ik := L_ik = A_ik L_kk^{−T}
        for j = k + 1 : i
            A_ij := A_ij − A_ik A_jk^T
        endfor
    endfor
endfor
```

Fig. 9. Algorithm-by-blocks for the Cholesky factorization. CHOL_BLK(B) refers to a blocked algorithm to compute the Cholesky factorization of B. On completion, this algorithm overwrites the lower triangular part of B with its Cholesky factor. The actual implementation is similar to those in Figures 2 (right) and 4, but for conciseness we present it as loops.

7.1 Cholesky Factorization of Banded Matrices

Consider a banded SPD matrix with bandwidth k_d, partitioned into $b \times b$ blocks as in (2) so that nonzero entries only appear in the diagonal blocks A_{kk}, the subdiagonal blocks $A_{k+1,k}, \ldots, A_{\min(N-1,k+D),k}$, and, given the symmetry of the matrix, $A_{k,k+1}, \ldots, A_{k,\min(N-1,k+D)}$. (If we assume for simplicity that $k_d + 1$ is an exact multiple of b then $D = (k_d + 1)/b - 1$.) An algorithm-by-blocks for the Cholesky factorization of this matrix is easily obtained from the algorithm in Figure 9 by changing the upper limit of the second loop to $\min(N - 1, k + D)$.

The ideas extend to provide algorithm-by-blocks for the LU factorization with incremental pivoting and the QR factorization of a banded matrix.

One of the advantages of using FLASH in the implementation of banded algorithm-by-blocks is that storage of a band matrix does not differ from that of a dense matrix. We can still view the matrix as a matrix of matrices but store only those blocks that contain nonzero entries into the structure. Thus, FLASH easily provides compact storage schemes for banded matrices.

For further details, see Quintana-Ortí et al. [2008c].

7.2 Inversion of SPD Matrices

Traditionally, the inversion of an SPD matrix A is performed as a sequence of three stages: compute the Cholesky factorization of the matrix $A = LL^T$; invert the Cholesky factor $L \rightarrow L^{-1}$; and form $A^{-1} := L^{-T}L^{-1}$. An algorithm-by-blocks has been given above for the first stage and two more algorithms-by-blocks can be easily formulated for the second and third stages. The result is an alternative algorithm-by-blocks that yields much higher performance than one which synchronizes all computation after each stage. For further details, see Chan et al. [2008].

The same approach provides an algorithm-by-blocks for the inversion of a general matrix via the LU factorization with incremental pivoting.

Bientinesi et al. [2008] showed it is possible to compute these three stages concurrently and that doing so enhances load-balance on distributed-memory architectures. Since the runtime system performs the operations on blocks out-of-order, no benefit results from a one-sweep algorithm.

8. EXPERIMENTAL RESULTS

In this section, we examine various multithreaded codes in order to assess the potential performance benefits offered by algorithms-by-blocks. All experiments were performed using double-precision floating-point arithmetic on two architectures:

—A ccNUMA SGI Altix 350 server consisting of eight nodes, each with two 1.5 GHz Intel Itanium2 processors, providing a total of 16 CPUs and a peak performance of 96 GFLOPS (96×10^9 floating-point operations/s). The nodes are connected via an SGI NUMAlink connection ring and collectively access 32 GB of general-purpose physical RAM, with 2 GB local to each node. Performance was measured by linking to the BLAS in Intel's Math Kernel Library (MKL) version 8.1.

—An SMP server with eight AMD Opteron processors, each one with two cores clocked at 2.2 GHz, providing a total of 16 cores and a peak performance of 70.4 GFLOPS. The cores in this platform share 64 GB of general-purpose physical RAM. Performance was measured by linking to the BLAS in Intel Math Kernel Library (MKL) version 9.1.

We report the performance of the following three parallel implementations in Figure 10:

—*LAPACK*: routines `dpotrf` (Cholesky factorization), `dgeqrf` (QR factorization), `dgetrf` (LU factorization with partial pivoting), and `dpbtrf` (Cholesky factorization of band matrices) in LAPACK 3.0 linked to multithreaded BLAS in MKL.

—*MKL*: multithreaded implementation of routines `dpotrf`, `dgeqrf`, and `dgetrf` in MKL.

—*AB*: our implementation of algorithm-by-blocks, with matrices stored hierarchically using the FLASH API, scheduled with the SuperMatrix runtime system and linked to serial BLAS in MKL. The OpenMP implementation provided by the Intel C compiler served as the underlying threading mechanism used by SuperMatrix on both platforms.

We consider the usual flop counts for the factorizations: $n^3/3$, $4n^3/3$, and $2n^3/3$, respectively, for the Cholesky, QR and LU factorizations of a square matrix of order n. The cost of the Cholesky factorization of a matrix with bandwidth k_d is computed as $n(k_d^2 + 3k_d)$ flops. Note that the algorithms-by-blocks for the QR and LU factorizations actually perform a slightly higher number of flops that represent a lower-order term in the overall cost.

When hand-tuning block sizes, an effort was made to determine the best values for all combinations of parallel implementations and BLAS. In the evaluation of the band factorization case, the dimension of the matrix was set to 5000. In this case, we report those results corresponding to the most favorable number of processors/cores for each implementation since using a lower number of resources in some cases resulted in a lower execution time.

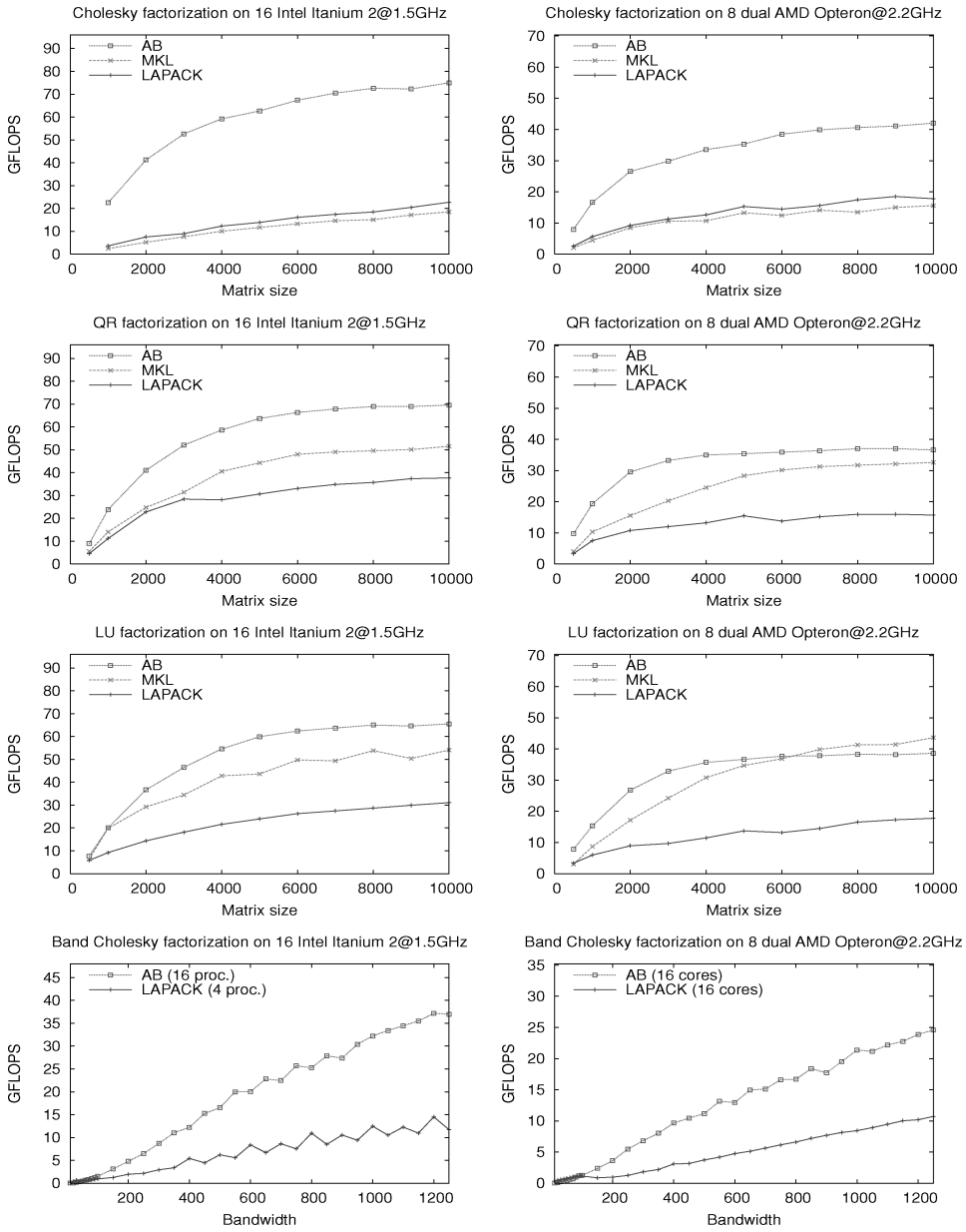


Fig. 10. Performance of the multithreaded factorization algorithms.

The results show that algorithms-by-blocks clearly outperform the codes in LAPACK and are competitive with highly tuned implementations provided by libraries such as MKL.

An interesting question is whether on multithreaded architectures it would be appropriate to instead use libraries such as ScaLAPACK or PLAPACK, which were designed for distributed-memory parallel architectures. In Bientinesi

et al. [2008], we presented evidence that, on multithreaded architectures, implementations that extract parallelism only within the BLAS outperform PLAPACK and ScaLAPACK, which use MPI [Gropp et al. 1994]. In this article, we show that extracting parallelism only within the BLAS is inferior to the proposed approach. Thus, by transitivity, the proposed approach can be expected to outperform MPI-based libraries like PLAPACK and ScaLAPACK on multithreaded architectures.

9. CONCLUSION

While architectural advances promise to deliver a high level of parallelism in the form of many-core platforms, we argue that it is programmability that will determine the success of these architectures. In this article, we have illustrated how the notation, APIs, and tools that are part of the FLAME project provide modern object-oriented abstractions to increase developer productivity and user-friendliness alike in the context of dense and banded linear algebra libraries. One of these abstractions targets multicore architectures by borrowing dynamic out-of-order scheduling techniques from sequential superscalar architectures. Results for the most significative (dense) matrix factorizations on two shared-memory parallel platforms consisting of a relatively large number of processors/cores illustrate the benefits of our approach.

The FLAME project strives to remain forward-looking. By maintaining a clean API design and clear separation of concerns, we streamline the process of taking a new algorithm from whiteboard concept to high-performance parallel implementation. The base FLAME/C API, the FLASH hierarchical matrix extension, and the SuperMatrix runtime scheduling and execution mechanism compliment each other through friendly abstractions that facilitate a striking increase in developer-level productivity as well as uncompromising end-user performance.

From the beginning, we have separated the SuperMatrix heuristic used for scheduling tasks from the library that implements the linear algebra operations. In Chan et al. [2007a], we demonstrated the benefits of using different heuristics to schedule suboperations to threads. As part of ongoing efforts, we continue to investigate the effects of different scheduling strategies on overall performance. We do not discuss this topic in the present article because our desire to focus the present article squarely on the issue of programmability. Another topic for future research is how to take advantage of the FLASH API's ability to capture multiple levels of hierarchy.

9.1 Additional Information

For additional information on FLAME visit

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

ACKNOWLEDGMENTS

We thank the other members of the FLAME team for their support. We thank John Gilbert and Vikram Aggarwal from the University of California at Santa Barbara for granting access to the NEUMANN platform.

REFERENCES

ADDISON, C., REN, Y., AND VAN WAVEREN, M. 2003. OpenMP issues arising in the development of parallel BLAS and LAPACK libraries. *Sci. Program. 11,* 2, 95–104.

AGARWAL, R. C. AND GUSTAVSON, F. G. 1989. Vector and parallel algorithms for Cholesky factorization on IBM 3090. In *SC '89: Proceedings of the ACM/IEEE Conference on Supercomputing*, New York. 225–233.

ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, L. S., DEMMEL, J., DONGARRA, J. J., CROZ, J. D., HAMMARLING, S., GREENBAUM, A., MCKENNEY, A., AND SORENSEN, D. 1999. *LAPACK Users' Guide*, 3rd Ed. Society for Industrial and Applied Mathematics, Philadelphia.

BALAY, S., BUSCHELMAN, K., EIJKHOUT, V., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., SMITH, B. F., AND ZHANG, H. 2004. PETSc users manual. Tech. rep. ANL-95/11—Revision 2.1.5, Argonne National Laboratory, Argonne.

BIENTINESI, P., GUNTER, B., AND VAN DE GEIJN, R. A. 2008. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw. 35*, 1, 1–22.

BIENTINESI, P., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2005. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Trans. Math. Softw. 31*, 1, 27–59.

BUTTARI, A., LANGOU, J., KURZAK, J., AND DONGARRA, J. 2007. A class of parallel tiled linear algebra algorithms for multicore architectures. LAPACK Working Note 191 UT-CS-07-600. University of Knoxville.

BUTTARI, A., LANGOU, J., KURZAK, J., AND DONGARRA, J. 2008. Parallel tiled QR factorization for multicore architectures. *Concurr. Computat. Pract. Experi. 20*, 13, 1573–1590.

CHAN, E., QUINTANA-ORTÍ, E. S., QUINTANA-ORTÍ, G., AND VAN DE GEIJN, R. 2007a. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, San Diego, 116–125.

CHAN, E., VAN ZEE, F. G., BIENTINESI, P., QUINTANA-ORTÍ, E. S., QUINTANA-ORTÍ, G., AND VAN DE GEIJN, R. 2008. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, 123–132.

CHAN, E., VAN ZEE, F. G., QUINTANA-ORTÍ, E. S., QUINTANA-ORTÍ, G., AND VAN DE GEIJN, R. 2007b. Satisfying your dependencies with SuperMatrix. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing* Austin, 91–99.

CHOI, J., DONGARRA, J. J., POZO, R., AND WALKER, D. W. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, 120–127.

COLLINS, T. AND BROWNE, J. C. 1995. Matrix++: An object-oriented environment for parallel high-perfomance matrix computations. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, Maui, 202–211.

DONGARRA, J. J., BUNCH, J. R., MOLER, C. B., AND STEWART, G. W. 1979. *LINPACK Users' Guide*. SIAM, Philadelphia.

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., AND DUFF, I. 1990. A set of level 3 basic linear algebra sub-programs. *ACM Trans. Math. Softw. 16*, 1, 1–17.

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

EDWARDS, H. C. AND VAN DE GEIJN, R. A. 2006. On application interfaces to parallel dense matrix libraries: Just let me solve my problem! FLAME Working Note #18 TR-2006-15. Department of Computer Sciences, University of Texas at Austin.

ELMROTH, E., GUSTAVSON, F., JONSSON, I., AND KAGSTROM, B. 2004. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Rev. 46*, 1, 3–45.

GOLUB, G. H. AND VAN LOAN, C. F. 1996. *Matrix Computations*, 3rd Ed. Johns Hopkins University Press, Baltimore.

GOTO, K. AND VAN DE GEIJN, R. A. 2008. Anatomy of a high-performance matrix multiplication. *ACM Trans. Math. Softw. 34*, 3, 1–25.

GROPP, W., LUSK, E., AND SKJELLUM, A. 1994. *Using MPI*. MIT Press, Cambridge.

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

GUNTER, B. C. AND VAN DE GEIJN, R. A. 2005. Parallel out-of-core computation and updating the QR factorization. *ACM Trans. Math. Softw. 31*, 1, 60–78.

GUO, J., BIKSHANDI, G., FRAGUELA, B., GARZARAN, M., AND PADUA, D. 2008. Programming with tiles. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, 111–122.

GUSTAVSON, F. G., KARLSSON, L., AND KAGSTROM, B. 2007. Three algorithms for Cholesky factorization on distributed memory using packed storage. In *Proceedings of the Workshop on State-of-the-Art in Scientific Computing*. Lecture Notes in Computer Science, vol. 4699. Springer, Berlin/Heidelberg, Germany, 550–559.

HERRERO, J. R. 2006. A framework for efficient execution of matrix computations. Ph.D. dissertation. Polytechnic University of Catalonia, Barcelona, Spain.

JOFFRAIN, T., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2004. Rapid development of high-performance out-of-core solvers. In *Proceedings of the Workshop on State-of-the-Art in Scientific Computing*. Lecture Notes in Computer Science, vol. 3732. Springer, Berlin/Heidelberg, Germany, 413–422.

KURZAK, J. AND DONGARRA, J. 2006. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. LAPACK Working Note 178 UT-CS-06-581. University of Tennessee, Knoxville.

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

LOW, T. M. AND VAN DE GEIJN, R. 2004. An API for manipulating matrices stored by blocks. FLAME Working Note #12 TR-2004-15. Department of Computer Sciences, University of Texas at Austin, Austin.

LU, H., COX, A. L., DWARKADAS, S., RAJAMONY, R., AND ZWAENEPOEL, W. 1997. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, 48–56.

MARKER, B. A., VAN ZEE, F. G., GOTO, K., QUINTANA-ORTÍ, G., AND VAN DE GEIJN, R. A. 2007. Toward scalable matrix multiply on multithreaded architectures. In *Proceedings of the 13th International European Conference on Parallel and Distributed Computing* (Rennes, France). 748–757.

NIEPLOCHA, J., HARRISON, R., AND LITTLEFIELD, R. 1996. Global arrays: A nonuniform memory access programming model for high-performance computers. *J. Supercomput. 10*, 2 (June), 197–220.

QUINTANA-ORTÍ, E. S. AND VAN DE GEIJN, R. 2009. Updating an LU factorization with pivoting. *ACM Trans. Math. Softw*.

QUINTANA-ORTÍ, G., QUINTANA-ORTÍ, E. S., CHAN, E., VAN DE GEIJN, R., AND VAN ZEE, F. G. 2008a. Design of scalable dense linear algebra libraries for multithreaded architectures: The LU factorization. In *Proceedings of the Workshop on Multithreaded Architectures and Applications*, Miami, 1–8.

QUINTANA-ORTÍ, G., QUINTANA-ORTÍ, E. S., CHAN, E., VAN ZEE, F. G., AND VAN DE GEIJN, R. A. 2008b. Scheduling of QR factorization algorithms on SMP and multi-core architectures. In *Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing* (Toulouse, France). 301–307.

QUINTANA-ORTÍ, G., QUINTANA-ORTÍ, E. S., REMÓN, A., AND VAN DE GEIJN, R. 2008c. An algorithm-by-blocks for SuperMatrix band Cholesky factorization. In *Proceedings of the 8th International Meeting on High-Performance Computing for Computational Science* (Toulouse, France). 1–13.

SKAGESTEIN, G. 1972. Rekursiv unterteilte matrizen sowie methoden zur erstellung von rechnerprogrammen fur ihre verarbeitung. Ph.D. dissertation. Universität Stuttgart, Stuttgart, Germany.

SORENSEN, D. C. 1985. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Trans. Comput. 34*, 3, 274–278.

STRAZDINS, P. 2001. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. *Int. J. Parall. Distrib. Syst. Netw. 4*, 1, 26–35.

TOLEDO, S. 1999. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms*, J. Abello and J. S. Vitter, Eds. American Mathematical Society, Boston, 161–179.

VALSALAM, V. AND SKJELLUM, A. 2002. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurr. Computat. Pract. Exper. 14*, 10, 805–840.

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

VAN DE GEIJN, R. A. AND QUINTANA-ORTÍ, E. S. 2008. *The Science of Programming Matrix Computations*. `www.lulu.com`.

VON HANXLEDEN, R., KENNEDY, K., KOELBEL, C. H., DAS, R., AND SALTZ, J. H. 1992. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 757. Springer, Berlin/Heidelberg, Germany, 97–111.

WISE, D. S., FRENS, J. D., GU, Y., AND ALEXANDER, G. A. 2001. Language support for Morton-order matrices. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, 24–33.

Received December 2007; revised July 2008, November 2008; accepted November 2008