A Systematic Approach to the Design and Analysis of Linear Algebra Algorithms

by

John Andrew Gunnels, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2001

Abstract

Over the last two decades, much progress has been made in the area of the high-performance sequential and parallel implementation of dense linear algebra operations. At what time can we confidently state that we truly understand this problem area and what form might evidence in support of this assertion take? It is our thesis that if we focus this question on the *software architecture* of libraries for dense linear algebra operations, we can claim to have reached the point where, for a restricted class of problems, we understand this area. In this dissertation, we provide evidence in support of this assertion by outlining a systematic and partially automated approach to the derivation and high-performance implementation of a large class of dense linear algebra operations.

We have arrived at a conclusion that the answer is to apply formal derivation techniques from Computing Science to the development of high-performance linear algebra libraries. The resulting approach has resulted in an aesthetically pleasing, coherent code that facilitates performance analysis, intelligent modularity, and the enforcement of program correctness via assertions. In this dissertation, we illustrate this observation by looking at the development of the Formal Linear Algebra Methods Environment (FLAME) for implementing linear algebra algorithms.

We believe that traditional methods of implementation do not reflect the natural manner in which an algorithm is either classified or derived. To remedy this discrepancy, we propose the use of a small set of abstractions that can be used to design and implement linear algebra algorithms in a simple and straightforward manner. These abstractions may be expressed in a script language that can be compiled into efficient executable code. We extend this approach to parallel implementations without adding substantial complexity.

It should also be possible to translate these scripts into analytical equations that reflect their performance profiles. These profiles may allow software designers to systematically optimize their algorithms for a given machine or to meet a particular resource goal. Given the more systematic approach to deriving and implementing algorithms that is facilitated by better abstraction and classification techniques, this sort of analysis can be shown to be systematically derivable and automated.

Contents

Chapter 1

Introduction

Our claim is that it is possible to create a system wherein one can code dense linear algebra routines in a very high-level, domain-specific language and still attain near-peak performance on distributed-memory parallel architectures. This dissertation provides evidence supporting this claim and describes the implications of such a system. Our thesis can be expressed as follows:

- We have discovered how to systematically derive a restricted class of linear algebra algorithms using formal derivation techniques.
- For this class of algorithms, compiler tools can be employed to reduce a domain-specific program to a list of operational requirements.
- In this domain, requirements can be paired to the functionality provided by a set of library routines if the annotations used to express those services are compatible with the requirements.
- For this class of algorithms, performance estimates of constructed routines can be made highly accurate if the underlying library is layered correctly and the language used to describe performance characteristics is suitably flexible.

The domain under study in this dissertation is restricted to a subset of dense linear algebra problems. This class includes the level-3 BLAS routines [25, 39], matrix factorization routines [44], and kernels involved in control theory [65, 64]. While this set of algorithms does not cover the gamut of dense linear algebra, it does comprise a useful, core set.

This chapter begins with an historical overview that summarizes the evolution of linear algebra software libraries. This is followed by a brief treatment of the insights that led us to the work presented here. We then explain how this work advances the state-of-the-art. After itemizing the contributions of our research, we present a summary of other research efforts whose goals are similar to our own. The final section of this chapter presents an outline of the dissertation.

1.1 Motivation

Advances in software engineering for scientific applications have often been led by techniques developed for libraries for dense linear algebra operations. The first such package to achieve widespread use and to embody new techniques in software engineering was EISPACK [68]. The mid-1970s witnessed the introduction of the Basic Linear Algebra Subprograms (BLAS) [55]. This version of the BLAS was a set of vector operations (now known as level-1 BLAS) that allowed libraries to attain high performance on computers possessing a flat memory while remaining portable between platforms. This library and its well-defined interface simultaneously enhanced code modularity and readability. The first successful library to exploit these BLAS was LINPACK [22].

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 operations in terms of level-2 (matrix-vector multiplication) and level-3 (matrix-matrix multiplication-like) BLAS operations [26, 25]. First released in the early 1990s, LAPACK [5] is a high-performance package for linear algebra operations. LAPACK is a portable library that provides a functionality that is a superset of both LINPACK and EISPACK. The LAPACK library heavily utilizes the level-3 BLAS and evinces high performance on essentially all sequential and shared-memory architectures.

A major simplification in the implementation of the level-3 BLAS stemmed from the observation that they can be cast in terms of optimized matrix-matrix multiplication [1, 47, 52]. The performance of the resulting libraries was comparable to that of the optimized, assembly-coded, vendor-supplied BLAS in many cases. Further, the implementations were more portable than previous BLAS libraries because they were written in Fortran. In those cases where the code was not performance transportable (i.e. where these BLAS did not compile into efficient assembly code), the ideas behind this research simplified the task of hand-coding the level-3 BLAS library.

With the advent of distributed-memory parallel architectures, LAPACK was no longer sufficient for the needs of high-performance scientific computing. LAPACK worked well with high-performance shared-memory systems, but was not written to be compatible with distributed-memory architectures. Distributed-memory architectures depend upon the applications and libraries to explicitly manage the physically distinct memories attached to the computational processors (nodes) of the system. Thus, a parallel version of LAPACK, ScaLAPACK [15], 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 analogous layers in the LAPACK software architecture. This decision was motivated by the fact that LAPACK had proven itself both robust and efficient. However, this decision complicated the implementation of ScaLAPACK. The introduction of data distribution across memories created a complication 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 difficult to maintain.

Recently, a number of projects have developed software for generating automatically tuned matrix-matrix multiplication kernels. These undertakings include the PHiPAC project [11] and the ATLAS project [76].

The PHiPAC research effort included a careful analysis of C implementations of matrix-matrix multiplication. By structuring the loops and memory references carefully, it is possible for a C compiler to generate highly efficient code for this algorithm. The PHiPAC research team produced a software system capable of generating efficient BLAS kernels through a generate-and-test strategy. This software generator created implementations of matrix multiplication algorithms that blocked matrices in every reasonable way. By executing these programs and monitoring the resulting performance, parameters for a high-performance matrix multiplication implementation could be determined.

The ATLAS project repackaged and simplified the methods developed in creating the PHiPAC system. In addition, the ATLAS system required less time to generate efficient linear algebra kernels. This efficiency was gained by avoiding PHiPAC's exhaustive search of the parameter space involved in determining optimal matrix blocking sizes. Unfortunately, as this search space was reduced through experience, not by a theoretical model, it is sometimes the case that ATLAS produces code with far less than optimal performance characteristics [42].

1.2 Our Approach

1.2.1 Recent Insights

The primary inspiration for much of the work presented in this dissertation came from our experience with the Parallel Linear Algebra Package (PLAPACK) [74]. PLAPACK achieves a functionality similar to that of ScaLAPACK, targeting the same distributed-memory architectures. In contrast to ScaLAPACK, PLAPACK uses an MPI-like [38] approach to hide indexing and data distribution details.

Work related to PLAPACK provided insights that motivated the approach presented in Chapter 2 and Chapter 3 of this document. Raising the level of abstraction at which one codes reduces the effort involved in implementing high-performance linear algebra library routines.

As we gained more experience with PLAPACK, a number of themes kept reappearing:

- The derivation of algorithms for different linear algebra operations was systematic.
- Similarly, the analysis of the resulting algorithms was systematic, although tedious and error-prone.
- For a given linear algebra operation, different algorithms provided better performance as the sizes of operands (matrices) changed [40]. This makes analysis necessary in order to be able to determine when and understand why different algorithms are superior.

We discovered that, in deriving algorithms for a new operation, we were applying formal derivation methods to the domain of algorithms for dense linear algebra operations. This led to our work on the Formal Linear Algebra Methods Environment (FLAME), research detailed in Chapter 2.

Linear algebra libraries are expected to contain routines that can deal with a broad range of operational tasks and to be written in a form that can be ported between different computational environments. The LAPACK library achieves both objectives by exploiting the BLAS. However, the use of libraries such as LAPACK has the disadvantages of requiring the applications programmer to perform time-consuming, involved, source code optimizations that are often not performance portable [50]. The work presented in Chapter 3 and Chapter 4 addresses this problem. By creating a language that allows the user to program at a level of abstraction higher than that of PLAPACK, little library knowledge is required of the programmer. An automated code generation system accepts programs written in this language and produces code that evinces superior performance on distributed-memory, parallel supercomputers. This is achieved by mechanically linking the high-level programs to a functionally-annotated version of the PLAPACK library.

A simple model of a distributed-memory parallel system is used for performance analysis in Chapter 5. This model reflects lessons learned while studying the issues related to the creation of high-performance matrix-matrix multiplication kernels for single processor machines with hierarchical memories [42]. This contrasts with code generation efforts such as PHiPAC and ATLAS, which employ brute force to search a parameter space for blocking sizes that accommodate multiple levels of memory hierarchy.

Together, these experiences and insights led us to conclude that for a subset of dense linear algebra operations, the derivation, implementation, and analysis of parallel algorithms is now a well-understood and systematic process.

1.2.2 A Solution: The Big Picture

The goal of linear algebra code production is to generate efficient code from a clear statement of mathematical requirements. Our strategy for achieving this objective is depicted in Figure 1.1. Specifically, it is our aim to replace the "Human Expert" of Figure 1.2, which reflects where previous research had led us, with systematic techniques and automated tools. The term "efficient" covers a number of sub-goals including reliability, speed, and transportability. These qualities are widely considered the primary value metrics of such computer codes. This dissertation targets the community of scientific library writers. Since one might safely suppose that these researchers are mathematicians or have strong mathematical backgrounds, the clear statement of mathematical requirements is a logical starting point. The mathematical specification of the problem must be known in order to generate code to solve that problem. In order to automate a system, this specification, represented by "A = LU" in Figure 1.1, must be made explicit.

The unified approach to the design and development of dense linear algebra algorithms that is presented in this document should be distinguished from the situation wherein development is *ad hoc*. When the development and tool sets are collected, not designed as

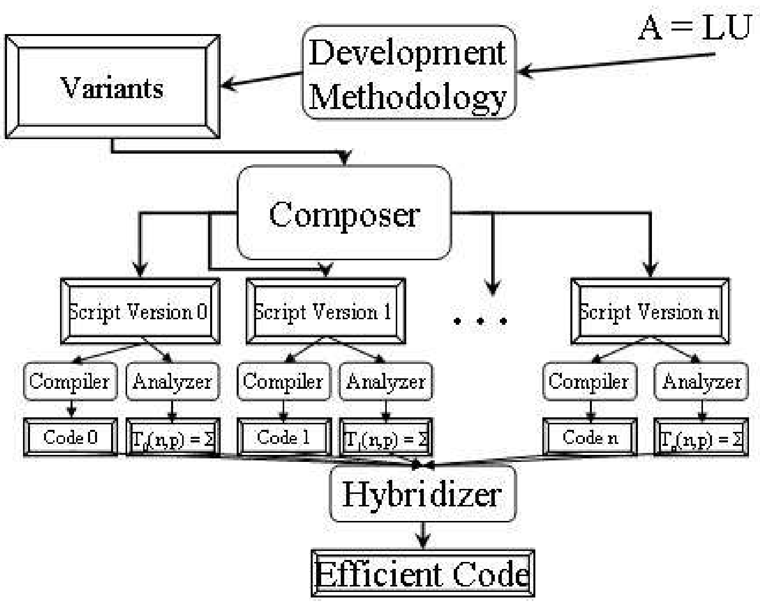


Figure 1.1: The Big Picture: As advanced in this dissertation

part of a holistic approach, they may supply as much baggage as leverage to a problem-solving environment.

Development Methodology

Given a mathematical specification of the problem, it is beneficial to have a consistent, methodological approach that enables one to construct an algorithm that satisfies this specification. If the approach is broadly applicable, it can be employed in the creation of the entire range of routines for a linear algebra library. If this methodology is systematic, it may be automated. In this dissertation, we present one such approach. FLAME is systematic in nature. In addition, FLAME can be utilized to generate a number of different algorithms, called *variants*, for the same mathematical problem specification.

Library Management: A Composer

One may create a number of variants corresponding to the same mathematical specification. In order to automate code generation, is useful to link together components that satisfy the same mathematical specification. In the work presented here, they are linked through annotations that expose the similarities in their functionality. This is the task of the “Composer.”

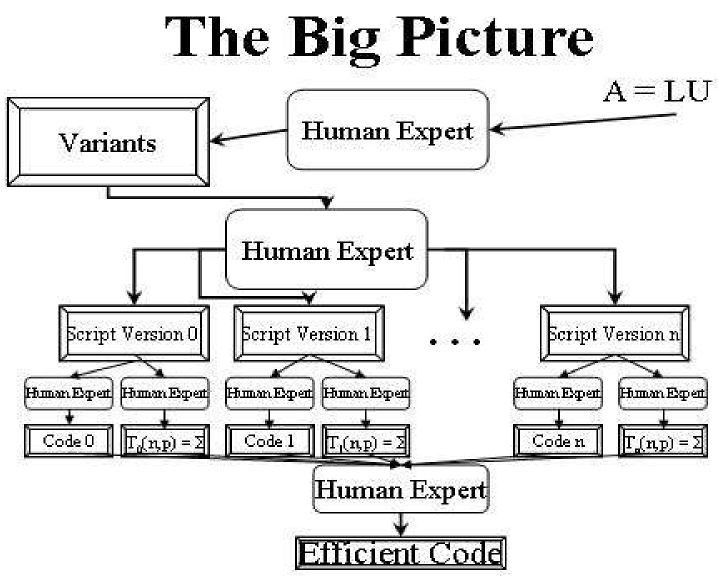


Figure 1.2: The Big Picture: As our research group has viewed it.

Input to the Composer is written in a high-level script language called PLAWright[1]. Scripts contain both an algorithmic component and the mathematical specification satisfied by that script. By annotating the scripts in this manner, the system can interchangeably use those scripts with the same functional characteristics.

It is a widely held belief that any automated system should allow for expert intervention. PLAWright, the language of the Composer, allows for hands-on modifications. These specializations take the form of such things as data distribution directives (in the context of parallel architectures), functional overrides (forcing the use a specific library call or code segment), and performance annotations (indicating the computational complexity of a component). In this dissertation, these specialized forms of a given variant are referred to as script *versions*. There is a single "vanilla," or plain, script corresponding to a variant constructed via FLAME, but there may be many specialized versions of that variant.

Code Generation and Analysis

Since the goal of the process under consideration involves the production of efficient code, we couple the code produced to an analysis procedure. By restricting our attention to the construction of code built on top of an existing library, the creation of such an analytical

[1] We would like to thank Sam Guyer for both the PLAWright name and a prototypical example of the language.

engine becomes a more precisely defined task.

Given a single script and a software library, there may be many ways to fulfill the requirements of the script with the services provided by the component library routines. It is often the case that different code instantiations exhibit different computational characteristics. It is also often true that no one routine is best for all situations. Differing operand dimensionalities and characteristics may make it necessary to dynamically select from many different routines in order to attain consistently near-optimal performance. This is called code hybridization. It makes sense to couple code generation and analysis in order to enable the production of hybridized code that is efficient across a wide range of problem instances. This dissertation work presents the PLANALYZER, a coupled code-production and code-analysis system.

The proof-of-concept implementation described in this dissertation limits the algorithmic area to a subset of dense linear algebra, the complexity measures to time, and the output language to C. However, this system can be extended to involve other complexity measures (such as memory usage) or to target other languages (such as Fortran).

1.3 Research Contributions

1.3.1 Systematizing Development

We have made systematic the derivation of a class of linear algebra algorithms through the use of simple formal derivation techniques. This advances the state-of-the-art by bringing formal derivation techniques to an area of software architecture that has made little use of them in the past. Our methodology is referred to as FLAME. Further, we have created a regimented structure for the expression of FLAME algorithms. This structure makes explicit the similarities and differences between closely related algorithmic variants. We have coupled this with the Formal Linear Algebra Methods Building Environment (FLAMBE)[2], which allows one to encode the routines in a form that mirrors the resultant FLAME algorithms.

FLAMBE code can handle matrix computations on both serial and parallel machines, with porting requiring only minor modifications. Thus, our work eases efforts required to construct a library that contains routines that share functionality, but address different levels of the memory hierarchy. This category of vertically integrated library is useful in high-performance, distributed-memory parallel computing.

1.3.2 Domain-Specific Languages

We have refined a domain specific language, called PLAWright, for the expression of dense linear algebra subroutines. We have also verified that algorithms expressed in this language can be compiled into code that executes on a parallel machine and into analytical code that reflects the complexity of the corresponding executable. Additionally, we have created a framework within which implicit assumptions regarding linear algebra algorithms are made

[2]This library has been referred to as FLAME in other documentation [41, 44].

explicit. Through PLAWright, we have created a language that allows for rapid prototyping and optimization, improving upon languages such as PLAPACK and MATLAB by raising the level of abstraction without sacrificing performance.

1.3.3 Automated Code and Analysis Generation

We have constructed an analytical model for homogeneous parallel computers that is simple, precise enough to meet our requirements, and applicable to modern microprocessors commonly used in the area of high-performance scientific computation. This modeling effort provided us with many insights into the design of a performance modeling language.

Our system allows an individual, who either lacks expert knowledge regarding the target architecture or the underlying libraries, to produce routines with admirable performance characteristics. The system we have created accomplishes this by utilizing expert knowledge, in the form of functional annotations, to construct a number of comparable programs from a single input script. In addition, this system is capable of analyzing the performance characteristics of these implementations in order to facilitate the selection of the best code available from the produced alternatives. Utilizing an analytical model represents an approach orthogonal to that of code generators such as PHiPAC and ATLAS.

1.4 Related Work: Integrated Systems

Below is a discussion of work related to "integrated systems" with goals similar to those addressed by the work in this dissertation. In subsequent chapters, the "Related Work" sections include research efforts that address the more narrow topic of that chapter.

1.4.1 MultiMATLAB

The MultiMATLAB project attempted to take advantage of a large existing code base and an integrated development environment [72]. The philosophy of the project was analogous to that underlying the ScaLAPACK project [15]. MultiMATLAB can utilize a number of MATLAB processes running on a set of processors. When coupled with a communications library, this enabled a parallel scripting environment. In this environment, a programmer can execute a script on the master processor and utilize the computational power of all of the processors in the system.

In contrast to MultiMATLAB, the system presented in this dissertation addresses the entire development process, from algorithmic development to code generation and analysis. Further, using our system results in code that exhibits admirable performance characteristics when executed on a distributed-memory, parallel supercomputer.

1.4.2 PSI

The PLAPACK-Server Interface (PSI) project [59] used an approach similar to that of MultiMATLAB[3]. Built on the PLAPACK library, the PSI package allows one to run scripts, written in MATLAB [58], Mathematica [77], or HiQ [17], on the master processor. These scripts can use the PLAPACK library to handle the requisite parallel computations while the system retains the ability to utilize the indicated computational environment in the case that:

- PLAPACK does not supply the desired functionality and
- The problem can fit on a single node.

Both MultiMATLAB and PSI allow the user to take advantage of the built-in graphics capabilities of the indicated commercial systems. The difference being that PSI can use the graphics capabilities of a single node while MultiMATLAB has the ability to utilize these graphics capabilities on all participating processors.

In contrast to PSI, our system allows the user to program at a level of abstraction that lies above that of the PLAPACK library. Further, unlike PSI, the research presented in this document includes algorithmic development and performance analysis.

1.4.3 FALCON

In sharp contrast to MultiMATLAB, the FALCON project [20, 57, 19] resulted in a system capable of compiling MATLAB code into an efficient parallel executable. It might appear that a large part of the work underlying the FALCON system was made obsolete by the compiler now available from the company that created MATLAB, The MathWorks^{TM}. However, this may not be the case. Parallel performance results are easy to get for the FALCON system while comparable figures for MultiMATLAB [63] are difficult to locate. However, it may be that the MultiMATLAB project is far more interested in flexibility than efficiency.

Unlike the FALCON project, our work addresses algorithmic development and, thus, presents an end-to-end development methodology.

1.4.4 Broadway

The Broadway Project at UT Austin is an effort to automatically optimize both software libraries and the applications that utilize them [51, 50]. This two-pronged approach is slightly different from the work presented in this dissertation. Broadway is primarily aimed at improving upon existing routines whereas the research thrust of this dissertation drops back to the creation of the algorithms and the use of a new language. Further, Broadway can be applied to libraries that do not involve scientific computation, whereas the PLANALYZER (see Chapters 3–5) is tied to that domain. Finally, our research takes a quantitative approach

[3]A tactic first utilized by STAR/MPI [16].

to the analysis and optimization of algorithms while Broadway's approach is qualitative in nature, as befits a more wide-ranging tool.

1.5 Overview of Dissertation

This overview is intended to serve to remind the reader of the components under study in this dissertation research. Each component builds upon the last, but no successor in the development process is entirely dependent upon its predecessor. The result is a system that has a "best of both worlds" flavor; the tools facilitate, but are not responsible for enabling, the next step in the process of development. The design methodology (FLAME) provides the underlying structure and philosophy for the rest of the system. The employment of FLAME results in algorithms of a specific structure. The next step in the process is the Composer, which utilizes the PLAWright language. The Composer accepts algorithms evincing this structure as input and may be used to specialize them before library linkage is performed. At that point, the PLANALYZER system is used to generate code and coupled analysis formulae through the use of an annotated library. Finally, the results of the PLANALYZER system can be used to create hybridized code.

1.5.1 Design: FLAME (Chapter 2)

The Formal Linear Algebra Methods Environment is a methodology that facilitates the systematic and formal derivation of dense linear algebra algorithms.

The FLAME methodology is built upon the use of loop invariants, a fundamental technique of computer science. While it is no surprise that this sort of methodology results in provably correct algorithms, the technique also allows for the creation of novel algorithms. There are many other benefits to this approach, and those are detailed in Chapter 2.

The systematic nature by which algorithms are derived with the FLAME philosophy is a strong indicator that this derivation process can be automated. Although such automation is *not* a part of the research presented in this dissertation, some evidence is offered in support of the assertion that FLAME can be partially mechanized. Mechanization of this step would result in an end-to-end, mechanized system for the creation of linear algebra libraries.

1.5.2 A Domain-Specific Language: PLAWright (Chapter 3)

Intimately tied to the derivation of the algorithms is the language in which one expresses the resulting artifact. An effort was made to allow the language of the algorithms to be virtually identical to the language of their implementation. FLAMBE is a step towards this goal, but it is not the final step, as Chapter 3, which introduces the PLAWright language, demonstrates.

1.5.3 Code Generation (Chapter 4)

In this text, the term "code generation" may be considered roughly synonymous with functional composition. Here, the central issue is linking to a library providing functional self-description via annotations. The approach used to mechanize linkage allows the different levels of the underlying library to be dealt with in a uniform manner.

The other desirable properties of an automated system, such as flexible library coupling, production code that reflects specializations in the high-level language, and high-performance codes based on little user direction, are also evident in the system examined in this document. Chapter 4 is concerned with functional linkage issues while Chapters 4 and 5 combine to deal with the automated production of high-performance code.

1.5.4 Performance (Chapter 5)

In the area of scientific computation, where linear algebra is a cornerstone, efficiency is crucial. In this chapter, we consider the issue of performance as it relates to algorithmic implementation. There are other interpretations of "performance" such as code creation time and the optimal use of the time and talent of human experts, but those are addressed elsewhere. The typical axes of quality in this field are the execution time and space required by executing routines.

Chapter 5 studies the issues pertinent to such concerns: modeling, evaluation, hybrid algorithms, and the performance annotations that enable the automation of this process.

1.5.5 Conclusion (Chapter 6)

Finally, a summary of the work and its contributions to the area of linear algebra library development is presented. Possible directions for further study and future work are also briefly discussed.

Chapter 2

Systematic Derivation of Variants

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 nigh impossible to upgrade.

Having struggled with these issues for more than a decade, we have concluded that the solution is to apply a technique from theoretical computer science, formal derivation, to the development of high-performance linear algebra libraries. We think that the resulting approach results in aesthetically pleasing, coherent code that 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 chapter, we illustrate our observations by looking at FLAME, a framework that facilitates the derivation and implementation of linear algebra algorithms.

2.1 Introduction

When considering the unmanageable complexity of computer systems, Dijkstra recently made the following observations [21]:

(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.

In this chapter, we make a number of contributions to the development linear algebra libraries. These contributions relate to the above observations as follows:

- 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 the correctness of the algorithms.
- By implementing the algorithms so that the code mirrors the algorithms that is the end-product of this derivation 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, FLAME, 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.

2.2 Overview

In Section 2.3.1 we review some basic insights from formal derivation theory. Next, in Section 2.4 we apply these insights to an illustrative example, LU factorization without pivoting, in order to develop a family of algorithms for a single, given operation. This is followed by Section 2.5, in which we summarize our systematic process for deriving linear algebra algorithms. Then, in Section 2.6 we show how library extensions added to the C programming language, together with careful formatting, allows 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 then demonstrated in Section 2.7. Performance is of concern in this area and in Section 2.8 we demonstrate that high performance is not compromised by raising the level of abstraction at which one codes. Finally, future directions and conclusions are given cursory treatment in Section 2.10 and a more in-depth look in Section 6.1.

2.3 Background

Some would immediately draw the conclusion that a change to a more modern programming language like C++ is at least 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 to the scientific computing community a programming model, object-based programming, that possesses many of the advantages typically associated with the intelligent use of an object-oriented language [69]. 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 the work presented here, 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 presented 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 [45, 67] and a low-level implementation of the sequential Basic Linear Algebra Subprograms (BLAS) [42, 44].

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. Some of our previous research [42] demonstrated 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 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 describe 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, previous work 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 [40]. We have come to 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) [45, 66]. 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 nigh impossible.

FLAME is the next 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 algorithm, its correctness can be asserted as well, and minimal debugging and testing is 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. The extent of this similarity can be seen by comparing Figure 2.3(a) and Figure 3.7

2.3.1 The Correctness of Loops

In a standard text by Gries and Schneider used to teach discrete mathematics to undergraduates in computer science we find the following material ([36], 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, although somewhat more informally, the correctness of a hybrid iterative/recursive LU factorization with partial pivoting in Section 2.7.

2.4 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.

2.4.1 A classical derivation

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

Partition

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

Now $A = LU$ translates to

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

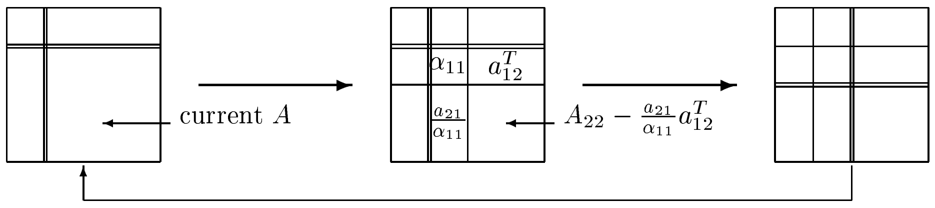
so the following equalities hold:

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

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).

The algorithm is usually implemented as a loop, as illustrated in Fig. 2.1. When presented in a classroom setting, this algorithm is typically accompanied by the following progression of pictures:



current A
α_{11}
a_{12}^T
$\frac{a_{21}}{\alpha_{11}}$
$A_{22} - \frac{a_{21}}{\alpha_{11}} a_{12}^T$

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 of the algorithm, and the implementation of the algorithm.

```
partition $A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$ where $A_{TL}$ is $0 \times 0$
do until $A_{BR}$ is $0 \times 0$
  repartition $\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & a_{01} & A_{02} \\ \hline\hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$ where $\alpha_{11}$ is a scalar

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

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

Figure 2.1: Unblocked lazy algorithm for LU factorization.

2.4.2 But what is the loop-invariant?

Notice that in the above algorithm the original matrix is overwritten by intermediate results until finally it 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 Fig. 2.1, $\hat{A}$ 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 of what the contents are at the beginning of the loop that implements the computation of the factorization (e.g., the loop in Fig. 2.1). To answer this question, partition the matrices as follows:

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

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 <u>T</u>op, <u>B</u>ottom, <u>L</u>eft, and <u>R</u>ight, respectively.

Notice that

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

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

so that the following equalities must hold:

$$A_{TL}^{(k)} = L_{TL}^{(k)} U_{TL}^{(k)} \tag{2.1}$$

$$A_{TR}^{(k)} = L_{TL}^{(k)} U_{TR}^{(k)} \tag{2.2}$$

$$A_{BL}^{(k)} = L_{BL}^{(k)} U_{TL}^{(k)} \tag{2.3}$$

$$A_{BR}^{(k)} = L_{BL}^{(k)} U_{TR}^{(k)} + L_{BR}^{(k)} U_{BR}^{(k)} \tag{2.4}$$

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 2.1–2.4.

Notice that in Equations 2.1–2.4 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 that 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 $A_{BR}^{(k)} - L_{BL}^{(k)} U_{TR}^{(k)}$.
- 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 2.1. The first and last conditions indicate that no computation has been performed or the final result has been computed, neither of which is a reasonable condition to maintain

Table 2.1: 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 (2.1) 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 (2.1) and (2.2) 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 (2.1) and (2.3) 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 (2.1), (2.2), and (2.3) have been satisfied.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline L_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right)$
(2.1), (2.2), and (2.3) have been satisfied and as much of (2.4) has been computed *without computing any part of* $L_{BR}^{(k)}$ *or* $U_{BR}^{(k)}$.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline L_{BL}^{(k)} & A_{BR}^{(k)} - L_{BL}^{(k)}U_{TR}^{(k)} \end{array}\right)$
(2.1), (2.2), (2.3), and (2.4) have all been satisfied.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline L_{BL}^{(k)} & L\backslash U_{BR}^{(k)} \end{array}\right)$

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 [23] 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 method 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 recent book on matrix factorization [71]. This comes as no surprise as 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 [43]. It needs to be evaluated against a large class of algorithms before we make any definitive claims regarding is usefulness.

2.4.3 Lazy algorithm

This algorithm is often referred to as a bordered algorithm in the literature. Stewart, [71] rather colorfully, refers to it as Sherman's march.

Unblocked Algorithm

Let us assume that only (2.1) has been satisfied. To determine the body of the loop (statement S), the question becomes how to update the contents of $\hat{A}$:

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

To answer this, repartition

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

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

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

After moving the double lines, the partitioning of A becomes

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

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

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

In summary, 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

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

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

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

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

  $a_{01} \leftarrow u_{01} = L_{00}^{-1} a_{01}$
  $a_{10}^T \leftarrow l_{10}^T = a_{10}^T U_{00}^{-1}$
  $\alpha_{11} \leftarrow \upsilon_{11} = \alpha_{11} - l_{10}^T u_{01}$

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

```
partition
  $A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$
do until $A_{BR}$ is $0 \times 0$
  determine block size $b$
  repartition
    $\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
      where $A_{11}$ is $b \times b$

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

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

Figure 2.2: Unblocked and blocked versions of the lazy variant for computing the LU factorization of a square matrix A (without pivoting).

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

$$\begin{array}{c||c|c} A_{00}^{(k)} = L_{00}^{(k)} U_{00}^{(k)} & a_{01}^{(k)} = L_{00}^{(k)} u_{01}^{(k)} & A_{02}^{(k)} = L_{00}^{(k)} U_{02}^{(k)} \\ \hline\hline a_{10}^{(k)T} = l_{10}^{(k)T} U_{00}^{(k)} & \alpha_{11}^{(k)} = l_{10}^{(k)T} u_{01}^{(k)} + \upsilon_{11}^{(k)} & a_{12}^{(k)T} = l_{10}^{(k)T} U_{02}^{(k)} + u_{12}^{(k)T} \\ \hline A_{20}^{(k)} = L_{20}^{(k)} U_{00}^{(k)} & a_{21}^{(k)} = L_{20}^{(k)} U_{01}^{(k)} + l_{21}^{(k)} \upsilon_{11}^{(k)} & A_{22}^{(k)} = L_{20}^{(k)} U_{02}^{(k)} + l_{21}^{(k)} u_{12}^{(k)T} + L_{22}^{(k)} U_{22}^{(k)} \end{array} \qquad (2.5)$$

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 Fig. 2.2 (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 1 *The algorithm in Fig. 2.2 (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

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

and the statement S is the body of the loop in Fig. 2.2 (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-presented algorithm. The derivation closely follows that of the unblocked algorithm: Again assume that only (2.1) 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

$$A = \left(\begin{array}{c||c} A_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline\hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c||c|c} A_{00}^{(k)} & A_{01}^{(k)} & A_{02}^{(k)} \\ \hline\hline A_{10}^{(k)} & A_{11}^{(k)} & A_{12}^{(k)} \\ \hline A_{20}^{(k)} & A_{21}^{(k)} & A_{22}^{(k)} \end{array}\right) \tag{2.6}$$

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

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

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

$$\begin{array}{l||l|l} A_{00}^{(k)} = L_{00}^{(k)}U_{00}^{(k)} & A_{01}^{(k)} = L_{00}^{(k)}U_{01}^{(k)} & A_{02}^{(k)} = L_{00}^{(k)}U_{02}^{(k)} \\ \hline\hline A_{10}^{(k)} = L_{10}^{(k)}U_{00}^{(k)} & A_{11}^{(k)} = L_{10}^{(k)}U_{01}^{(k)} + L_{11}^{(k)}U_{11}^{(k)} & A_{12}^{(k)} = L_{10}^{(k)}U_{02}^{(k)} + L_{11}^{(k)}U_{12}^{(k)} \\ \hline A_{20}^{(k)} = L_{20}^{(k)}U_{00}^{(k)} & A_{21}^{(k)} = L_{20}^{(k)}U_{01}^{(k)} + L_{21}^{(k)}U_{11}^{(k)} & A_{22}^{(k)} = L_{20}^{(k)}U_{02}^{(k)} + L_{21}^{(k)}U_{12}^{(k)} + L_{22}^{(k)}U_{22}^{(k)} \end{array} \qquad (2.7)$$

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 Fig. 2.2 (right) and Fig. 2.3(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 1.

Theorem 2 *The algorithm in Fig. 2.2 (right) overwrites a given non-singular, $n \times n$ matrix, A, with its LU factorization.*

2.4.4 Row-lazy algorithm

As a point of reference, Stewart [71] calls this algorithm Pickett's charge south.

Let us assume that only (2.1) and (2.2) 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 (2.6). Our assumption is that $\hat{A}_k$ holds

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

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

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

Thus, it suffices to compute $L_{10}^{(k)}$, $L\backslash U_{11}^{(k)}$, and $U_{12}^{(k)}$. Recalling the equalities in (2.7) 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 Fig. 2.3(c).

The proof of the following theorem is similar to that of Theorem 1.

Theorem 3 *The algorithm in Fig. 2.3(c) overwrites a given non-singular, $n \times n$ matrix, A, with its LU factorization.*

2.4.5 Column-lazy algorithm

This algorithm is referred to as a left-looking algorithm in [27] while Stewart [71] calls it Pickett's charge east.

Let us assume that only (2.1) and (2.3) 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 Fig. 2.3 (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 1.

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

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

  (d) Column-lazy:                            (e) Row-column-lazy:
  View $A_{00}$ as $L\backslash U_{00}$                 $A_{11} \leftarrow L\backslash U_{11} = \mathrm{LU}(A_{11} - L_{10} U_{01})$
  $A_{01} \leftarrow U_{01} = L_{00}^{-1} A_{01}$               $A_{12} \leftarrow U_{12} = L_{11}^{-1}(A_{12} - L_{10} U_{02})$
  $A_{11} \leftarrow L\backslash U_{11} = \mathrm{LU}(A_{11} - L_{10} U_{01})$   $A_{21} \leftarrow L_{21} = (A_{21} - L_{20} U_{01}) U_{11}^{-1}$
  $A_{21} \leftarrow L_{21} = (A_{21} - L_{20} U_{01}) U_{11}^{-1}$

  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
```

Figure 2.3: LU factorization without pivoting for five commonly encountered variants.

Theorem 4 *The algorithm in Fig. 2.3(d) overwrites a given non-singular, $n \times n$ matrix, A, with its LU factorization.*

2.4.6 Row-column-lazy algorithm

This algorithm is often referred to as Crout's methods in the literature [18].

We assume that only (2.1), (2.2), and (2.3) 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 Fig. 2.3 (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 1.

Theorem 5 *The algorithm in Fig. 2.3(e) overwrites a given non-singular, $n \times n$ matrix, A, with its LU factorization.*

2.4.7 Eager algorithm

This algorithm is often referred to as classical Gaussian elimination.

We proceed under the assumption that (2.1), (2.2), and (2.3) have been satisfied, and as much of (2.4) 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 (2.6). Notice, our assumption is that $\hat{A}_k$ holds

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

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

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

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 (2.7) 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 Fig. 2.3(a). Notice that this algorithm is the blocked equivalent to the algorithm derived in Section 2.4.1.

The proof of the following theorem is similar to that of Theorem 1.

Theorem 6 *The algorithm in Fig. 2.3(a) overwrites a given non-singular, $n \times n$ matrix, A, with its LU factorization.*

2.5 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, as 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 of the algorithm. While we use 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 [43].

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

2.6 Encoding the Algorithm in C

In this section we briefly discuss how dense linear algebra algorithms, as presented in Figs. 2.1–2.3, 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 and our FLAME framework represents a refinement of this methodology.

2.6.1 Classic implementation with the BLAS

Let us consider the blocked eager algorithm for the LU factorization presented in Fig. 2.3 (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 Fig. 2.4: 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 Fig. 2.5.

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

2.6.2 The algorithm *is* the code

We would argue that it is relatively easy to generate the code in Fig. 2.5 given the algorithm in Fig. 2.3(a) and the picture in Fig. 2.4. 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 is not much of a problem if one only had to implement just one algorithm, real difficulties may arise when implementing a number

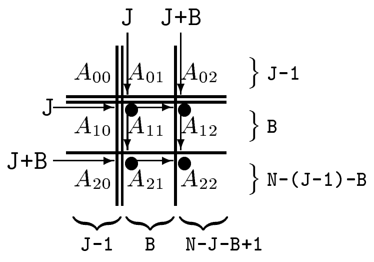


Figure 2.4: Partitioning of matrix A with all dimensions annotated when $A_{00} = A_{TL}$ is $(j-1) \times (j-1)$.

```
  SUBROUTINE LU_EAGER_LEVEL3( N, A, LDA, NB )
  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
```

Figure 2.5: Fortran implementation of blocked eager LU factorization algorithm using the BLAS. (Find the bug without referring to Fig. 2.4 or the text!)

```
Partition $A = \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$
do until $A_{BR}$ is $0 \times 0$
    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 $b \times b$
    ------------------------------------------
        insert update here
    ------------------------------------------
    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
```

Figure 2.6: Algorithm skeleton for LU factorization without pivoting.

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 Fig. 2.3, 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 Figs. 2.6 and 2.7, 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 Fig. 2.7, implement the different variants of the LU factorization:

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);
```

```c
#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 );
  }
}
```

Figure 2.7: A code skeleton for the C implementation of many of the blocked LU factorization algorithms using FLAME.

Row-lazy algorithm

```
FLA_Trsm(FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
         ONE, A00, A10);
FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A01, ONE, A11);
FLA_LU_nopivot_level2(A11);
FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A02, ONE, A12);
FLA_Trsm(FLA_LEFT,  FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
         ONE, A11, A12);
```

Column-lazy algorithm

```
FLA_Trsm(FLA_LEFT,  FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
         ONE, A00, A01);
FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A01, ONE, A11);
FLA_LU_nopivot_level2(A11);
FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A20, A01, ONE, A21);
FLA_Trsm(FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
         ONE, A11, A21);
```

Row-column-lazy algorithm

```
FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A01, ONE, A11);
FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A20, A01, ONE, A21);
FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A02, ONE, A12);
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);
```

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);
```

2.6.3 Positive features of the FLAME approach

Naturally, 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 Fig. 2.5? 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 Fig. 2.3. However, translating those to code like that given in Fig. 2.5 would require *several* careful considerations of the picture in Fig. 2.4. Moreover, due to the detailed and extensive indexing involved in that approach, considerable testing would be required before one could declare the code bug-free. By contrast, given the algorithms, it has been argued that generating all variants using FLAME is straightforward. 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_`.

2.6.4 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.

2.6.5 Proving the implementation correct

In Section 2.4.3 we proved correctness of the lazy algorithm and in subsequent subsections of Section 2.4 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 are 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 to 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.

Naturally, 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 is easily automatable because of the one-to-one relation between the two, so that a very common mistake, namely the code not reflecting the algorithm (when one considers a textual version of the algorithm as it might be presented in a textbook), can be obviated. A second common mistake made with such codes, indexing errors, is eliminated from the top-level expression of FLAME code because FLAME does no 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 believe that the *coupled* system leads to programs whose correctness follows from a mathematical derivation of the algorithm.

2.7 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.

2.7.1 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

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

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 $P_m(p_0, p_1, \cdots, p_{k-1})A$ equals the matrix that results after swapping rows 0 and p_0 followed by swapping rows 1 and $p_1 + 1$, etc., 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 $p_{i+1}+1$, etc., in that order.

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

$$P_{m;0:n-1}A = LU \tag{2.8}$$

2.7.2 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 2.8 is equivalent to

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

or

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

where

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

Partitioning

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

we find that

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

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 2.1–2.4 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 2.1, 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 2.1 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 2.1, 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.

2.7.3 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 $\left(\begin{array}{c} \bar{L}_{10}^{(k)} \\ \hline \bar{L}_{20}^{(k)} \end{array}\right) \leftarrow Q_1 \left(\begin{array}{c} \tilde{L}_{10}^{(k)} \\ \hline \tilde{L}_{20}^{(k)} \end{array}\right)$. Note that $L\backslash U_{11}^{(k)}$ and $L_{21}^{(k)}$ are defined by Equation 2.9, 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

$$\left(\begin{array}{c} \hat{A}_{11}^{(k)} \\ \hline \hat{A}_{21}^{(k)} \end{array}\right) = \left(\begin{array}{c} L_{11}^{(k)} \\ \hline \bar{L}_{21}^{(k)} \end{array}\right) U_{11}^{(k)} \tag{2.9}$$

  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: $\left(\begin{array}{c} \hat{A}_{10}^{(k)} \\ \hline \hat{A}_{20}^{(k)} \end{array}\right) \leftarrow Q_1 \left(\begin{array}{c} \tilde{L}_{10}^{(k)} \\ \hline \tilde{L}_{20}^{(k)} \end{array}\right)$.

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

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

```
Partition $A = \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$ and $p = \left(\begin{array}{c} p_T \\ \hline\hline p_B \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$ and $p_T$ has 0 elements
do until $A_{BR}$ is $0 \times 0$
    Determine block size $b$
    Partition
        $\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
            where $A_{11}$ is $b \times b$
    Partition
        $\left(\begin{array}{c} p_T \\ \hline\hline p_B \end{array}\right) = \left(\begin{array}{c} p_0 \\ \hline\hline p_1 \\ \hline p_2 \end{array}\right)$
            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 $b$.
    ----------------------------------------------------------------
    $\left[ A_{BR}^{(1)}, p_1 \right] \leftarrow \left[ \left( \begin{array}{c} L \backslash U_{11} \\ \hline L_{21} \end{array} \right), p_1 \right] = \mathrm{LU}_{\mathrm{piv}}(A_{BR}^{(1)})$
    $A_{BL} \leftarrow P(p_1) A_{BL}$
    $A_{BR}^{(2)} \leftarrow P(p_1) A_{BR}^{(2)}$
    $A_{12} \leftarrow U_{12} = L_{11}^{-1} A_{12}$
    $A_{22} \leftarrow A_{22} - L_{21} U_{12}$
    ----------------------------------------------------------------
    Continue with
        $\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$
        $\left(\begin{array}{c} p_T \\ \hline\hline p_B \end{array}\right) = \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline\hline p_2 \end{array}\right)$
enddo
```

Figure 2.8: Eager blocked LU factorization with partial pivoting.

```
void FLA_LU( FLA_Obj A, FLA_Obj ipiv, int nb_alg )
{
  < declarations >

  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                   /* ************** */
                     &ABL, /**/ &ABR,
                 /* with */ 0, /* by */ 0, /* submatrix */ FLA_TL );
  FLA_Part_2x1( ipiv, &ipivT,
                    /* ****** */
                       &ipivB,
                 /* with */ 0, /* length submatrix */ FLA_TOP );

  while (b = min(min( FLA_Obj_length(ABR), FLA_Obj_width(ABR)), nb_alg))
  {
    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 );
    FLA_Repart_2x1_to_3x1( ipivT,                 &ipiv0,
                         /* ***** */             /* ***** */
                                                  &ipiv1,
                            ipivB,                &ipiv2,
                 /* with */ b, /* length ipiv1 split from */ FLA_BOTTOM );
    FLA_Part_1x2( ABR,   &ABR_1, &ABR_2,
                 /* with */ b, /* width submatrix */ FLA_LEFT );
/***************************************************************************/

    if ( nb_alg <= 4 ) FLA_LU_level2(ABR_1, ipiv1);
    else               FLA_LU       (ABR_1, ipiv1, nb_alg/2);

    FLA_Apply_pivots(FLA_SIDE_LEFT, FLA_NO_TRANSPOSE, ipiv1, ABL);
    FLA_Apply_pivots(FLA_SIDE_LEFT, FLA_NO_TRANSPOSE, ipiv1, ABR_2);
    FLA_Trsm(FLA_SIDE_LEFT, FLA_LOWER_TRIANGULAR,
             FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
             ONE, A11, A12);
    FLA_Gemm(FLA_NO_TRANSPOSE,FLA_NO_TRANSPOSE, MINUS_ONE,A21,A12,ONE,A22);

/***************************************************************************/
    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 );
    FLA_Cont_with_3x1_to_2x1( &ipivT,                  ipiv0,
                                                       ipiv1,
                             /* ***** */             /* ***** */
                              &ipivB,                  ipiv2,
                 /* with ipiv1 added to */ FLA_TOP );
  }
}
```

Figure 2.9: FLAME recursive LU factorization with partial pivoting.

In Fig. 2.8 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

$$\left[A_{BR}^{(1)}, p_1\right] \leftarrow \left[\left(\frac{L\backslash U_{11}}{L_{21}}\right), p_1\right] = \mathrm{LU}_{\mathrm{piv}}(A_{BR}^{(1)}) \tag{2.10}$$

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 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 2.10 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 $\left[A_{BR}^{(1)}\right]_{p_1}$ before the swap.

2.7.4 Implementation

A FLAME implementation of the blocked algorithm in Fig. 2.8 is given in Fig. 2.9. Notice that a FLAME implementation of the unblocked algorithm would look similar. Let us assume that the latter is correctly implemented in the FLAME routine

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

Now, the correctness of algorithm in Fig. 2.8 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 [49, 29].

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

Figure 2.10: Level-3 BLAS operations implemented as part of the productivity experiment.

2.8 Experiments

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

2.8.1 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 Fig. 2.10 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 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 Figs. 2.5 and 2.7. 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.

2.8.2 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 2.5, 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 experiments can be found in [43].

2.8.3 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 Fig. 2.9 followed by forward and backward substitution, i.e., essentially the LINPACK benchmark. For comparison, we also measured the performance of the equivalent operations provided by ATLAS R3.2 [76].

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 Fig. 2.11 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 [42]. 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.)

`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.

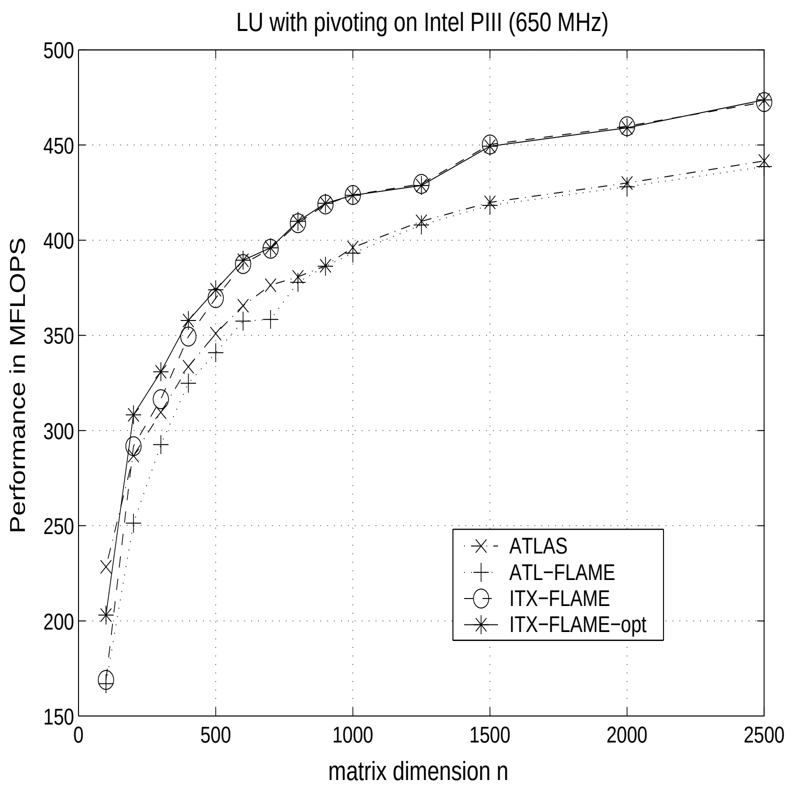


Figure 2.11: Performance of LU factorization with pivoting followed by forward and backward substitution.

Notice that for small matrices the performance of `ATL-FLAME` is somewhat inferior to that of ATLAS, due to 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.

2.9 Related Work

Libraries for dense linear algebra operations have often led advances in software engineering for scientific applications. The first such package to achieve widespread use and to embody new techniques in software engineering was EISPACK [68]. 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) [55]. At that time, the BLAS were a set of vector operations that allowed libraries to attain high performance on vector supercomputers while remaining highly portable between platforms, simultaneously enhancing modularity and code readability. The first successful library to exploit these BLAS was LINPACK [22]. By the mid-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 operations in terms of matrix-matrix multiplication-like operations, the level-3 BLAS [25]. LAPACK [5], 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 super set 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 [1, 47, 52]. 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 [15], 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 are 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. Our own 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 [74]. Indeed, the primary inspiration for FLAME came from PLAPACK.

A number of recent efforts have explored the notion of utilizing hierarchical data structures for storing matrices [4, 46, 48]. 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 algorithms that exploit this data structure, 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 FLAMBE can be extended to accommodate more complex data structures in the context of hierarchical memories.

2.10 Chapter Summary

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 chapter, 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 is a direct translation of the resulting algorithm, 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.

Let us again examine the observations of Dijkstra:

> (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.

In this chapter, and through years of experience writing parallel linear algebra libraries, we have learned this lesson the hard way. While a large percentage of code and an even larger percentage of effort was devoted to the development of test code for packages like LAPACK and ScaLAPACK, we believe that the more formal and systematic approach that underlies FLAMBE and PLAPACK has reduced the need for such testing, while simultaneously increasing our confidence in the implementation.

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

Notice that our approach carefully derives the program, making the proof of its correctness an inherent part of its derivation.

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

Since the code reflects the algorithm, the argument that the algorithm is correct carries over to an argument that the code is correct.

Throughout this chapter we have focused 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 automatically generates stable algorithms, we do claim that the skeleton used to express the algorithm, and to implement the code, can be used to implement known algorithms with known numerical stability properties. It also facilitates the discovery and implementation of new algorithms for which numerical properties can then be subsequently established.

Chapter 3

From Variant to Multiple Versions

This chapter introduces a coding environment that allows the user to implement algorithms in a higher level language than FLAMBE (seen in Chapter 2). This language, dubbed "PLAWright," is the interface to the automated system, the PLANALYZER, discussed in the next three chapters of this dissertation. The reader is referred to Figure 3.1. In this figure, the automated components of this dissertation are depicted in an abbreviated form. This chapter focuses on, the "High-level Program," which is to be input.

Subsequent chapters demonstrate that this programming approach does not require one to forsake performance considerations when moving to a computational environment. In this chapter, the focus is on the high level of abstraction in programming which frees the user from many low-level concerns. This allows the programmer to utilize algorithms that bear the promise of increased performance, but might have been overlooked because of the required investment in programming, debugging, and maintenance time and effort [7].

3.1 Motivation

There are a number of reasons to adopt the coding style delineated in this chapter. Some of those motivating factors present themselves in the context of sequential systems while others are made apparent only when distributed computational environments are considered. The issues and difficulties associated with traditional approaches are discussed here along with an overview of the solution advocated in this work.

3.1.1 Coding Matrix Algorithms: The Sequential World

There are two traditional strategies for coding sequential matrix algorithms:

1. Simple indexing into the original array [22] and

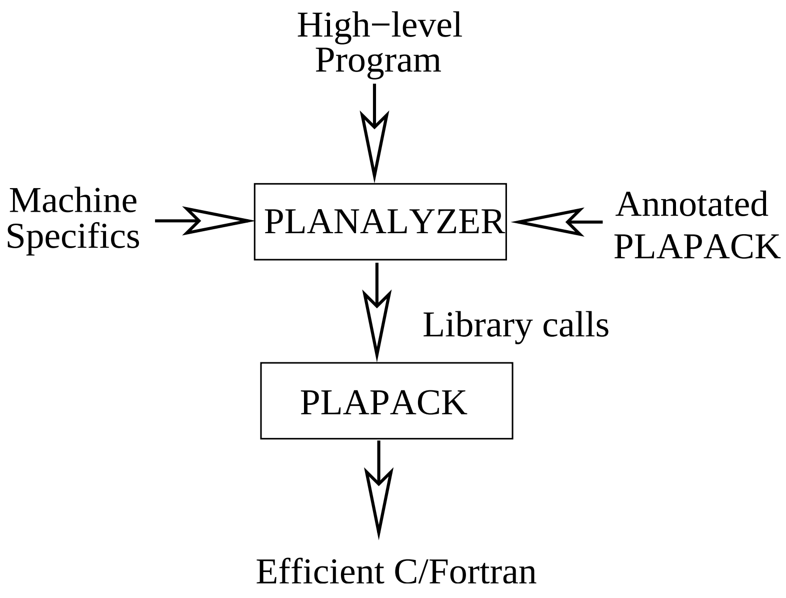


Figure 3.1: Overview of the PLANALYZER

2. Indexing combined with a standard library supplying computational kernels [5] such as the Basic Linear Algebra Subprograms (BLAS) [26, 25].

Problems with Traditional Approaches

As has been mentioned, both of these approaches share the same shortcomings. Both approaches require that one keep track of where in the matrices the computations are occurring. The amount of bookkeeping required to do this as algorithms become more sophisticated is daunting and error-prone. In order to avoid mounting design and maintenance costs, algorithms that are more ambitious are often abandoned for this reason.

Notice that the original derivation of these algorithms does not involve these indices. It is the attempt to mesh two ways of viewing matrices that appears to cause the problem.

3.1.2 Coding Matrix Algorithms: Extending to Parallel

Traditionally, extending a library [15] or an integrated development environment [72] to a parallel environment has involved the goal of maximizing code re-use. Some newer software systems [19] appear to view this goal as secondary and they provide some tools for the integration of alien modules.

In contrast, software systems with a more coherent "vision," such as PETSc [9] and PLAPACK [74], take a more unified view of the computational environment and present the user with a library that has a more consistent interface. These libraries also avoid the pitfall of hiding parallelism in order to avoid added complexity. They expose levels of parallelism to the algorithmic designer in a flexible manner [8].

Problems with Traditional Approaches

While the re-use of existing library components is a laudable goal, we think that it is unnecessarily constricting. For example, the ScaLAPACK project [15] attempts to make maximal re-use of LAPACK [5] components. This approach forces one to view parallel computational systems as vastly more complex than sequential systems. While it is true that such architectures are somewhat more complicated, it is the adaptation of sequential libraries to parallel environments that causes many programming errors. The troublesome artifacts of such adaptation include lengthening parameter lists and poorly documented interactions between levels of both hardware and software.

The second error that can be seen in the design of some of these software packages is an unfortunate coupling of computation and communication libraries. An example is ScaLAPACK's *initial* coupling with the Basic Linear Algebra Communication Subprograms (BLACS) [6] routines. While part of the problem rested in the non-modular nature of such a tightly-coupled arrangement, a more profound penalty is incurred by the limited breadth of abstraction. Some communications patterns that are not supported by the BLACS library arise naturally in parallel linear algebra routines. An example is the BLACS library's inability to redistribute an $n \times 1$ matrix object across the entire processor grid (i.e. view the grid as a linear processor array). This operation is often important for load-balance in linear algebra solver algorithms [28].

3.1.3 Proposed Solution

If the source of the problem is the interaction between design systems and abstraction sets that are incompatible, it makes sense to eliminate this conflict. The development of an abstraction set that reflects the derivation of the algorithms can minimize the severity of this conflict.

The proposed solution for addressing the difficulties in the parallel environment is to couple the philosophy of libraries, such as PLAPACK, with the ease of programming available in environments such as the one provided by MATLAB [58]. This allows the user to exploit or insulate themselves from the details of the parallel programming environment. Allowing the user to code in this manner is not only easier on the user, but allows the user to implement algorithms that are more sophisticated.

Chapter 2 demonstrated that the goal of coupling the design system and the abstraction set available to the implementor is achievable using conventional languages. Given the initial derivation, and the problems expounded above, it seems that many of the problems encountered could be obviated if one were allowed to code in a format such as the one depicted in Figure 3.2. The same script may be translated into code that operates on a single processor or into code that operates on multiple processors. In this case, the efficiency of the resulting code relies on the sophistication of the translator and the underlying library. In addition, as is discussed in Section 3.1.4, the same software system allows the user to implement both other variants (Figure 3.3) of the algorithm as well as specialized versions (Figure 3.4) while programming in the same style.

```
L has_property unit_lower_triangular ; // (* Permanent Property *)
U has_property upper_triangular ;      // (* Same as non-unit *)
A has_property square ; // (* Actually, Square here *)
L === A ; // {Recursive} {Permanent}  (* Establish name equivalence *)
U === A ; // {Recursive} {Permanent}
partition A =>    / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  where ATL is 0 by 0 ;
do until ABR is 0 by 0
   partition      / ATL # ATR \
                  |###########|
                  \ ABL # ABR /
                                   =>    / A00 # A01 | A02 \
                                         |#################|
                                         | A10 # A11 | A12 |
                                         |-----#-----------|
                                         \ A20 # A21 | A22 /
                  where A11 is local           and
                        A11 is locally square  and
                        A11 is nb by nb ; // No larger than is implied
  A01 = U01 <- L00^-1 * A01 ;
  A10 = L10 <- A10 * U00^-1 ;
  A11 = (L11\U11) <- A11 - L10 * U01 ;
  A11 = (L11\U11) <- lu_fact(A11)   ;
  partition
                  / ATL # ATR \
                  |###########|
                  \ ABL # ABR /   <=     / A00 | A01 #  A02 \
                                         |------------------|
                                         | A10 | A11 #  A12 |
                                         |##################|
                                         \ A20 | A21 #  A22 / ;
enddo;
L =!= A;
U =!= A;
```

Figure 3.2: Computer-readable script for Lazy version of LU factorization

Notice that both Figure 3.3 and Figure 3.4 illustrate the executable form of the Eager version of the LU decomposition. While both figures correspond to the algorithm presented in Figure 2.3 (a) on page 27, the latter is not a “vanilla” form of the variant. It is what I refer to as a *version* of that variant; in this case, the version is only slightly specialized. This version contains a directive intended to result in data locality in a distributed-memory computational environment. A discussion regarding the import of such specializations is delayed until Chapter 4.

```
L has_property unit_lower_triangular ; // (* Permanent Property *)
U has_property upper_triangular ;
A has_property square ;
L === A ; // {Recursive} {Permanent}  (* Establish name equivalence *)
U === A ; // {Recursive} {Permanent}
partition A =>    / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  where ATL is 0 by 0 ;
do until ABR is 0 by 0
   partition      / ATL # ATR \
                  |###########|
                  \ ABL # ABR /
                                  =>     / A00 # A01 | A02 \
                                         |#################|
                                         | A10 # A11 | A12 |
                                         |-----#-----------|
                                         \ A20 # A21 | A22 /
                  where A11 is nb by nb ; // No larger than is implied
   A11 = (L11\U11) <- lu_fact(A11) ;
   A12 = U12 <- L11^-1 * A12 ;
   A21 = L21 <- A21 * U11^-1 ;
   A22 <-  A22 -  L21 * U12  ;
   partition
                  / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  <=      / A00 | A01 #  A02 \
                                         |------------------|
                                         | A10 | A11 #  A12 |
                                         |##################|
                                         \ A20 | A21 #  A22 / ;
enddo;
L =!= A;
U =!= A;
```

Figure 3.3: Computer-readable/PLAWright-compilable script for the Eager variant of LU factorization

```
L has_property unit_lower_triangular ; // (* Permanent Property *)
U has_property      upper_triangular ;
A has_property square ; // (* Actually, Square here *)
L === A ; // {Recursive} {Permanent}  (* Establish name equivalence *)
U === A ; // {Recursive} {Permanent}
partition A =>    / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  where ATL is 0 by 0 ;
do until ABR is 0 by 0
   partition      / ATL # ATR \
                  |###########|
                  \ ABL # ABR /
                                  =>     / A00 # A01 | A02 \
                                         |#################|
                                         | A10 # A11 | A12 |
                                         |-----#-----------|
                                         \ A20 # A21 | A22 /
                  where A11 is local           and
                        A11 is locally square  and
                        A11 is nb by nb ; // No larger than is implied

   A11 = (L11\U11) <- lu_fact(A11) ;
   A12 = U12 <- L11^-1 * A12 ;
   A21 = L21 <- A21 * U11^-1 ;
   A22 <-  A22 -  L21 * U12  ;
   partition
                  / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  <=      / A00 | A01 #  A02 \
                                         |------------------|
                                         | A10 | A11 #  A12 |
                                         |##################|
                                         \ A20 | A21 #  A22 / ;
enddo;
L =!= A;
U =!= A;
```

Figure 3.4: Script for Eager version of parallel LU factorization

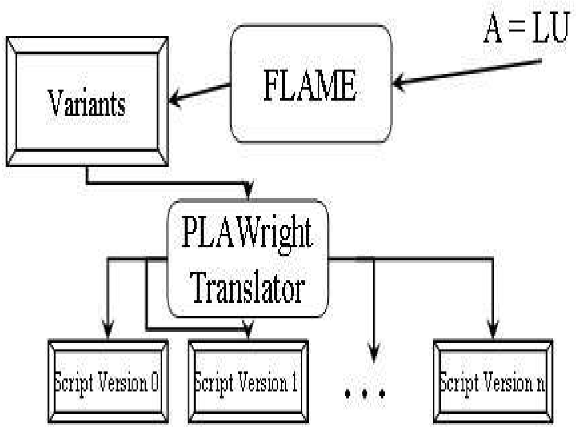


Figure 3.5: Where PLAWright fits into the "grand scheme" of things.

3.1.4 Where PLAWright Fits In

Let us reconsider Figure 1.1, the "Big Picture" illustrated on page 5 of Chapter 1. While the FLAME development methodology is systematic, it is not automated. Therefore, there is something of a cognitive break between FLAME and the remainder of the programming environment discussed in this dissertation. After the variants are produced by the FLAME methodology, the process is entirely mechanized. The PLAWright Composer marks the point of demarcation between systematization and mechanization.

Automation is desirable in this area because it allows the programmer to focus their efforts on creating algorithms instead of translating these algorithms into code. PLAWright allows the user to produce versions of the different coding variants (see Figure 3.5, ahead). The language also serves to enforce some level of programming discipline. This discipline comes about because the language of the scripts has a syntax that can be expressed in terms of a context-free-grammar (CFG). In our implementation, the CFG is encoded in the language of the ANTLR [61, 62] compiler tool.

3.2 Issues

There are a number of considerations that affect the design of a domain-specific language. The language should capture the central abstractions involved in the domain, retain some level of flexibility and extensibility, and be of a form that can be automatically translated into an executable. In this section we discuss these issues.

3.2.1 Abstraction

Ease-of-use is an important property in a linear algebra library. Unfortunately, this property has often been either ignored or relegated to a position of minor importance. On the one hand, the reason for this is simple and not, entirely, incorrect: performance is important. People do not use a "friendly" application library for code-development if its performance characteristics are unacceptably poor. On the other hand, people like to use such programming environments (e.g. MATLAB) for proof-of-concept designs. Therefore, it makes sense to utilize multiple levels of abstraction in a mathematical library.

Such levels, optimally, present a somewhat unified interface to the library user. However, it is often the case that different levels in such a library cannot be completely congruent [15] in that they cannot all take the same arguments or argument types. Nonetheless, it is usually possible to present the user with understandable "variations on a theme" in these cases *if* one starts with a systematic approach to the entire library.

Why Level Consistency Is Important

An important component of the systematic approach that enables this consistency between programming layers lies in the devising of a set of useful abstractions to describe the algorithms under consideration. Selecting the right abstractions gives one the ability to express algorithms in a compact and understandable manner. Further, it allows for a consistent vocabulary when discussing algorithms at various levels of detail.

Important Concepts

Because this dissertation largely ignores issues of memory hierarchy until Chapter 5 (see page 64), it should come as no surprise that there are few general abstractions involved in designing dense linear algebra algorithms. Only three appear necessary for our purposes. Object manipulation and (data) component computation are required in the previously presented algorithms. Object property transformations are somewhat hidden, but are also necessary. Here, the terms object and component have different meanings. An object includes both the data component and the other properties of the operand (e.g. size). The component is the raw data on which mathematical operations are performed. The manipulation of and computation on objects influences the corresponding properties of those objects. Thus, one could consider a computation to involve the entire object. The problem with this view is that the property computations are of a very different nature than the data computations. Further, the data computations are well understood; while, traditionally, the property transforms have been either ignored or made almost entirely implicit. Chapter 4, which deals with automatic code generation (and specialization) presents a case for making these property transformations explicit.

For a concrete example that involves these issues, let us consider the Eager variant of LU-decomposition that is illustrated in Figure 3.6:

The entire algorithm relies upon two things:

```
partition $A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
        where $A_{TL}$ is $0 \times 0$
do until $A_{BR}$ is $0 \times 0$
    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 nb by nb
    $A_{11} \leftarrow$ LU fact.$(A_{11})$
    $A_{12} \leftarrow U_{12} = L_{11}^{-1} A_{12}$
    $A_{21} \leftarrow L_{21} = A_{21} U_{11}^{-1}$
    $A_{22} \leftarrow A_{22} - L_{21} U_{12}$
    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
```

Figure 3.6: Eager approach to LU factorization (paraphrased)

1. How to delimit a block of an operand and
2. The manner in which these operand blocks interact

I think that this figure depicts a natural way to express such an algorithm. However, as I restrict the programming environment to the ASCII domain, the goal of this work is to allow the input form to match that illustrated in Figure 3.3. The following sections demonstrate how this goal can be achieved in an implementation.

Object Manipulation Linear algebra routines typically involve matrices, vectors, and scalars. The number of operands involved in an algorithm depends upon the algorithm under consideration. The "nature" of such objects includes their instantiation and individuality. For example, in the LU decomposition there are conceptually three objects, all matrices, A, L, and U. A is instantiated (has size, values etc.) when routine begins, while L and U are not. Also, while we may consider the three matrices to be distinct entities for the purposes of deriving equations, the algorithms shown in Chapter 2 were composed under the restriction that L and U overwrote A as the algorithms progressed. This co-location of data influences the manner in which algorithms are constructed.

These issues motivate all of the object manipulation primitives that are required for the subset of dense linear algebra algorithms under consideration in this document. The rest of this section examines the manipulators needed. Although object properties, such as being lower-triangular, may be affected by both manipulation and computation they are considered a separate abstraction and not as a facet of either the other two nor as an emergent artifact of their interaction.

The first abstraction needed is *co-reference* to an existing object. While other manipulations can be "abused" to yield this operation, our goal here is not the construction of a minimal set of primitives, but the creation of a small and useful set. The need of this operation occurs at the very beginning of the LU decomposition algorithm and is related to the previously discussed co-location property. The LU algorithm begins with a single matrix, A, that is to be factored into two matrices L and U. Because L and U eventually occupy the same space as A, the logical thing to do is to view A as sharing components with L and U.

The next manipulator to be considered is the one that performs *(re-)partitioning*. After we have all of the objects that we need to carry out the algorithm, we need to be able to refer to different subsets of the objects. In the cases presented in this dissertation, the situation is even simpler, as we wish to be able to "name" only *contiguous* parts of the *data components* of the objects under consideration. Because we may begin with a two-dimensional matrix and wish to consider a two-dimensional submatrix of the same object, it seems that two abstractions are required: splitting the object vertically and splitting the object horizontally. In addition to the direction of the split, the size of the resulting object would also need to be specified in the realization of this abstraction. Further, if a matrix can be decomposed through splitting, we should also have the ability to combine parts of a matrix, or vector, in order to create a new object.

Clarification and Justification There are some unanswered questions regarding the abstractions given above. Some of these ambiguities involve the issue of co-reference. The final question concerns the direction of assignment involved in each type of abstraction.

In Figure 3.3 we forced L to co-reference A. This has the same outcome as splitting A into some number of objects where all but one of the objects has a nil size (0×0). While such a split is valid, there is a drawback to this approach: it does not match the *algorithms* as they were presented in Chapter 2. Also, while it is true that the algorithms could be rewritten to use this "zero-split" co-reference, it is our contention that this would be somewhat less intuitive than the alternative.

Another co-reference ambiguity involves the scope of the operations and conditions. Consider that we state that L co-refers to (the lower-triangular part of) A. While it does not arise in the presented algorithm, we may later wish to partition L in one way and A in another. The language used must provide some way to distinguish between permanent and temporary co-references. In PLAWright, the syntactic distinction involves the use of `===` to indicate a "locked," recursive equality and `==` to indicate a temporary equality.

Issues regarding the "direction" of assignments must also be considered. For example, if we were to employ `A = L` notation, it would be apparent that A was being assigned to L, as L was assumed to be non-instantiated. In order to make the semantics of the language unambiguous in this regard there are at least three possibilities:

1. Rely on input specifications to indicate which objects are initialized.

2. Use positional queues. For example, in C, the line `X = Y;` unambiguously means that Y should be assigned to X. **or**

3. Use operational queues (e.g. Y $\Rightarrow$ X and X $\Leftarrow$ Y would both assign Y to X).

The first option is undesirable because assignment may involve two initialized objects, or, in the case of an assigment that involved composition, *groups* of objects. Therefore, we eliminate the first option from consideration. There seems to be no compelling reason to favor either of the other two conventions. While the third allows smaller syntactic alterations to the algorithmic description to disambiguate the meaning of the code, one might reasonably argue that the second alternative yields cleaner code. In any case, we adopt the third alternative as the convention in this dissertation and allow `=` to serve as something of a comment that can be used as an assertion of operand compatibility.

Computation Only three computational operators are required by the software system discussed in this document. All three were used in the different example derivations of the LU decomposition algorithm: multiplication, (triangular) inversion, and addition.

In a linear algebra library, one must expect to perform some form of *matrix multiplication*. This may be a matrix-matrix, a matrix-vector, or a vector-vector multiplication. For the moment, let us only consider the cases that are well-defined. That is, in the case where we wish to determine the value of $A \times B$, A is of size $m \times k$ and B is of size $k \times n$. In this case, the primitive used corresponds to the standard matrix-matrix multiplication algorithm.

There are other cases that must be considered. The first such case arises when the operation is apparently not well-defined but one of the operands is a scalar (a 1×1 matrix). This operation needs its own semantics to determine if a given calculation is well-defined. Such an operation is considered well-defined if the objects involved are initialized. The other cases that must be considered are the result of matrix properties: matrix structure and transposition status.

A linear algebra object may have many applicable structural specifiers. However, only upper- and lower-triangular matrices are considered in this document. In both cases, only *part* of the matrix is considered to be defined. Operations involving such objects must never refer to (read or write) the undefined portion of the objects.

Matrix inversion is often required in linear algebra. In the Eager LU decomposition algorithm presented in this dissertation, it is used to determine A_{12} where $A_{12} = L_{11}^{-1}A_{12}$, for instance.

As matrix structure has been considered in this section, it should be pointed out that the matrix inversion required for the LU algorithm(s) presented here is of a restricted type: the inversion of a triangular matrix. As a practical matter, true inversion would not be performed due to the special structure of the matrix under consideration. Instead, the operation would be implemented as a computationally less expensive triangular solve. The details are unimportant. The situation is highlighted simply because it is an illustration of the distinction between abstraction and implementation.

The last two operators, *matrix addition* and *matrix subtraction*, are so similar that, given the scalar multiplication discussed above, only one is required. However, it is easier to discuss the algorithms when both are used, so both are included. Both operations are well-defined when both operands are of equal dimensions and have the same structure.

Property Manipulations While one may think that the concepts of object manipulation and computation have some overlap, this is not the case in this dissertation. Consider the preceding sections. Manipulation involved a single data component while computation referred to object interaction. The barrier between abstraction classes becomes somewhat more difficult to draw when one considers object properties.

As has been mentioned, properties could be considered as facets of both manipulation and computation. For reasons already discussed, there are benefits to viewing them as separate entities. However, even with this point-of-view in mind, we must not lose sight of the fact that both manipulations and computations can affect object properties. Similarly, properties can affect manipulations and computations.

While there are many *potential* object properties, we consider only a few. In this document, there are only two properties that we consider when dealing with objects: *size* and *shape*.

The size property specifies the dimensions of the object under consideration. This property can be used for a number of things. Most fundamentally, it can be used during the interaction of two objects to determine if the proposed interaction is well-defined.

Shape properties can be used for the same purpose. Here, we consider only a few possible shape (perhaps more properly called "constituency") categories. Among these are: full, empty, zero, and triangular. Empty is essentially the same as unspecified and the "other half" of a triangular object is treated as unspecified (uninitialized) during all computational interactions.

There are also properties that may not be properly attached to any one object. For example, we have already discussed the idea of a co-reference object manipulation (i.e. establishing object equivalence). Co-referencing can be viewed as a one- or two-way relationship. If we view it as a one-way relationship, one object is "secondary" and the property may be attached to either object. However, if the relationship is considered to be two-way, there are two choices:

1. The property can be attached to both objects or

2. The objects can be attached to their mutual relationship

We adopt the view that the relationship is two-way and the property is attached to both objects.

Finally, there is the *transposition* property to consider. This property indicates whether an object exists in the transposed state, or if an object is equivalent to the transpose of a second object (often, a "parent" object). While this may arise from a transposition operation (a manipulation operation not previously considered), they are different things precisely because the property can be attached to an object or deleted from that object's properties regardless of its "true" state. The transposed state changes the applicability of the computational manipulators in the expected way.

Iterators and Selectors Iteration and selection are required in any mathematical programming language. The PLAWright language uses only one iterator :

`do until <condition>/enddo`.

Similarly, there is only one selector:

`if<condition>-then-else`,

a construct that resembles the C or Pascal if-then(-else) operator.

In PLAWright, there is a restriction as to what these `<condition>`s may contain. As it is now implemented, the condition must be related to the properties mentioned above (structure and size).

3.2.2 A Domain-Specific Language for Linear Algebra

The language presented in this chapter is intended to mirror the algorithms produced when employing the FLAME methodology and to allow one to realize, in code, the abstractions discussed in Section 3.2.1. Largely, it does so successfully, but the disparities between FLAME and PLAWright deserve a bit of exposition. Similarly, as the previous chapter maintained that the FLAMBE coding style enabled code and algorithm to be virtually indistinguishable, the claims made there must be reconsidered.

FLAME vs. PLAWright

In an attempt to allow the novice to create programs with efficiencies that are close to those produced by an expert, the first step is to allow the novice to program in an environment that only requires knowledge of standard linear algebra symbols and a few easily-remembered notational conventions.

Figure 3.3 on page 52 illustrates the simple, "executable" format of the Eager version of the LU decomposition.

There are few differences between this script and the corresponding algorithm presented in the previous chapter. The similarity of the two is primarily the result of the fact that the abstractions were designed around this style of presentation. We would also maintain that this style of presentation is a "natural" one and, optimally, the code should conform closely to it. The differences between the two are primarily the result of the fact that there are a number of implicit assumptions that a human makes or "figures out;" our compilation system makes no such assumptions.

The most obvious difference is the ASCII-ized nature of the PLAWright language. This dissimilarity exists because standard compiler technology does not easily lend itself to programming in or interpreting PostScript, the standard form of output for technical papers.

Another notable difference stems from the need to add certain properties (via annotations) to the co-reference status that needs to be maintained between L, U, and A. While it is clear that these names are all to initially refer (in some sense) to the same object, it is not necessarily the case that this property is to be inherited by all named sub-objects (recursive) or that the property is never voided (permanent). Because FLAMBE was written to respect C and Fortran, this idea of explicit co-reference appeared to be at odds with the philosophy of the language. The reader may have noticed that an analogous disparity exists between FLAME and PLAWright, but was not mentioned in Section 3.2.2. In FLAME the co-reference remains implicit; only in PLAWright does it seem to present itself as a natural part of the language.

Another disparity involves the addition of ";" (semicolons) to the end of each command in the PLAWright language. This was done for reasons of expediency; statement separators tend to make things clearer to translators without having a profound impact on the readability of the script. They may even make the script somewhat easier to read in the absence of the formatting imposed on Figure 3.3, as whitespace is unimportant to the PLAWright-compiler. This practice also tends to allow for the generation of more informative error messages, since statement and line numbers have unambiguous meaning in this case.

Finally, the reader may have noted the transposition of `=` and `<-` between the algorithms and the scripts. This was done intentionally in order to point out that such things are often a matter of taste and the compilation system can be altered to suit such differences with simple symbol (token) renaming. Here, we have taken ease-of-programming a step further and extended the goal to ease of language extension. Since the implementation of the language relies upon ANTLR compiler technology, allowing such customization seemed necessary and proved to be simple to perform.

PLAWright vs. FLAMBE

The PLAWright implementation of Eager LU factorization is depicted in Figure 3.3. This Figure bears a strong resemblance to Figure 2.3(a). By way of contrast, let us consider the expression of the eager LU algorithm as expressed using the FLAMBE system as is seen in Figure 3.7. Great pains have been taken to make the FLAMBE language resemble FLAME's language of algorithmic expression. However, the confines of the C programming language necessitated some of the lexical distance between the two expressive forms. By adding the appropriate comments, as is done in Figure 3.8, one can make the purpose of the code more readily evident. However, the use the PLAWright domain-specific language obviates the need for such comments. The comments in the FLAMBE code (Figure 3.8) are virtually identical to the corresponding lines in the PLAWright script (Figure 3.3).

Because performance is a consideration, it should be pointed out that the use of such a script language does not require one to sacrifice their quest for stellar performance. In this chapter, the manner in which the user can specialize the scripts so as to achieve superior

```

void PLA_LU_eager( PLA_Obj A, int nb );
{
  < declarations >
  PLA_Create_constants_conf_to( A, &minus_one, NULL, &one );
  PLA_Obj_partition_4( A,          &ATL, /**/ &ATR,
                                 /* ************** */
                                   &ABL, /**/ &ABR,
       /* with */ 0, /* by */ 0, /* submatrix */ PLA_SUBMATRIX_TL );
  while ( size = PLA_OBJ_GLOBAL_LENGTH( ABR ) ){
    b = min( size, nb );
    PLA_Obj_repartition_4_to_9( ATL, /**/ ATR,         &A00, /**/ &A01, &A02,
                              /* ************ */      /* ****************** */
                                     /**/              &A10, /**/ &A11, &A12,
                                ABL, /**/ ABR,         &A20, /**/ &A21, &A22,
         /* with */ b, /* by */ b, /* A11 split from submatrix */ PLA_SUBMATRIX_BR );
    PLA_LU_level2( A11 );
    PLA_Trsm( PLA_SIDE_LEFT, PLA_LOWER_TRIANGULAR,
              PLA_NO_TRANSPOSE, PLA_UNIT_DIAG,
              one, A11, A12 );
    PLA_Trsm( PLA_SIDE_RIGHT, PLA_UPPER_TRIANGULAR,
              PLA_NO_TRANSPOSE, PLA_NONUNIT_DIAG,
              one, A11, A21 );
    PLA_Gemm( PLA_NO_TRANSPOSE, PLA_NO_TRANSPOSE,
              minus_one, A21, A12, one, A22 );
    PLA_Obj_continue_with_9_to_4( &ATL, /**/ &ATR,         A00, A01, /**/ A02,
                                        /**/               A10, A11, /**/ A12,
                                /* ************** */    /* ****************** */
                                  &ABL, /**/ &ABR,         A20, A21, /**/ A22,
         /* with A11 added to submatrix */ PLA_SUBMATRIX_TL );
  }
  < cleanup >
}
```

Figure 3.7: FLAMBE (parallel C version) code for the Eager version of LU factorization

```

void PLA_LU_eager( PLA_Obj A, int nb );
{
  < declarations >
  PLA_Create_constants_conf_to( A, &minus_one, NULL, &one );
  PLA_Obj_partition_4( A,          &ATL, /**/ &ATR,
                                 /* ************** */
                                   &ABL, /**/ &ABR,
       /* with */ 0, /* by */ 0, /* submatrix */ PLA_SUBMATRIX_TL );
  while ( size = PLA_OBJ_GLOBAL_LENGTH( ABR ) ){
    b = min( size, nb );                              /* Determine block size b        */
    PLA_Obj_repartition_4_to_9( ATL, /**/ ATR,         &A00, /**/ &A01, &A02,
                              /* ************ */      /* ****************** */
                                     /**/              &A10, /**/ &A11, &A12,
                                ABL, /**/ ABR,         &A20, /**/ &A21, &A22,
         /* with */ b, /* by */ b, /* A11 split from submatrix */ PLA_SUBMATRIX_BR );
    PLA_LU_level2( A11 );                             /* A11 <- L\U11 = LU fact( A11 ) */
    PLA_Trsm( PLA_SIDE_LEFT, PLA_LOWER_TRIANGULAR,    /* A12 <- U12 = inv(L11) * A12   */
              PLA_NO_TRANSPOSE, PLA_UNIT_DIAG,
              one, A11, A12 );
    PLA_Trsm( PLA_SIDE_RIGHT, PLA_UPPER_TRIANGULAR,   /* A21 <- L21 = A21 * inv(U11)   */
              PLA_NO_TRANSPOSE, PLA_NONUNIT_DIAG,
              one, A11, A21 );
    PLA_Gemm( PLA_NO_TRANSPOSE, PLA_NO_TRANSPOSE,     /* A22 <- A22 - A21 * A12        */
              minus_one, A21, A12, one, A22 );
    PLA_Obj_continue_with_9_to_4( &ATL, /**/ &ATR,         A00, A01, /**/ A02,
                                        /**/               A10, A11, /**/ A12,
                                /* ************** */    /* ****************** */
                                  &ABL, /**/ &ABR,         A20, A21, /**/ A22,
         /* with A11 added to submatrix */ PLA_SUBMATRIX_TL );
  }
  < cleanup >
}
```

Figure 3.8: Commented FLAMBE (parallel C version) code for the Eager version of LU factorization

performance is addressed, while the discussion regarding the effects of these specializations will be largely delayed until Chapter 5.

3.2.3 Parallel Specializations and Extensions

Thus far, details regarding computational environments have been largely glossed over. The different approaches were described in a manner that avoided any real consideration of a computational environment even if the text occasionally used the term "sequential" to supply a basis for communication. While this is appropriate if one wishes to treat the presented derivation methods as useful educational tools, it falls short if one wishes to bring these ideas to fruition in the real world.

To realize the presented algorithms and to implement the primitives discussed thus far is a straightforward task *if* the developer is restricted to the monolithic memory model [60]. However, to extend the algorithms so that they are efficient in a distributed-memory system requires more work.

This subsection presents a number of issues that only arise in the parallel architectural arena and show that few changes are required to extend the algorithms and abstractions already presented so as to comply with the restrictions and requirements imposed by this model.

Why Specialization Is Important

When one shifts one's focus from the abstract environment of algorithmic derivation to that of implementation, a number of issues arise. In the arena of linear algebra algorithms, these concerns can largely be pared down to one: memory hierarchy considerations. For example, in the parallel architecture case there are two basic programming paradigms (models): shared-memory and distributed-memory. In this document the focus is on an approach that was designed with distributed-memory machines in mind, but with the ability to treat the underlying architecture as if it were based on the shared-memory model. The reason for this approach is simple; it is desirable to accommodate both models and, since the shared-memory model offers much less control than the distributed model, using a strictly shared-memory model would prove sub-optimal from a performance point-of-view [75].

The primary advantage of the shared-memory model is programming ease. Most of the examples in this dissertation, and all those presented thus far, could remain unchanged if they were to be implemented on a shared-memory machine. The reason for this is simple; shared-memory models treat a computational system, whether it has non-uniform memory architecture (NUMA) characteristics or not, as if it were a "UMA" architecture. Unfortunately, ignoring the NUMA nature of a system can result in sub-optimal performance. By layering the abstractions and the library derived from those abstractions so as to ease transition from a shared view to a distributed view, the user is allowed to trade convenience for performance in a flexible manner. In Chapter 5 we demonstrate how this design philosophy also allows for the implementation of a (simple) performance analyzer that can dynamically analyze the trade-offs as the user transitions between approaches.

Writing Parallel Algorithms

There are two ways to view the construction of parallel algorithms in this setting. For simplicity, let us call them “hands-off” and “hands-on.” Both philosophies have potential advantages . . . and disadvantages.

The hands-off approach is to rely upon the underlying computational environment to deal with issues related to parallelism. This, of course, requires that the underlying code translation and instantiation mechanism be capable of treating the computational environment as a shared-memory system. Figure 3.9 shows how the code for the parallel version of Eager LU decomposition might appear in such a script.

```
L has_property unit_lower_triangular ; // (* Permanent Property *)
U has_property      upper_triangular ;
A has_property square ; // (* Actually, Square here *)
L === A ; // {Recursive} {Permanent}  (* Establish name equivalence *)
U === A ; // {Recursive} {Permanent}
partition A =>    / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  where ATL is 0 by 0 ;
do until ABR is 0 by 0
   partition      / ATL # ATR \
                  |###########|
                  \ ABL # ABR /
                                  =>     / A00 # A01 | A02 \
                                         |#################|
                                         | A10 # A11 | A12 |
                                         |-----#-----------|
                                         \ A20 # A21 | A22 /
                  where A11 is local           and
                        A11 is locally square  and
                        A11 is nb by nb ; // No larger than is implied

   A11 = (L11\U11) <- lu_fact(A11) ;
   A12 = U12 <- L11^-1 * A12 ;
   A21 = L21 <- A21 * U11^-1 ;
   A22 <-  A22 -  L21 * U12  ;
   partition
                  / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  <=      / A00 | A01 #  A02 \
                                         |------------------|
                                         | A10 | A11 #  A12 |
                                         |##################|
                                         \ A20 | A21 #  A22 / ;
enddo;
L =!= A;
U =!= A;
```

Figure 3.9: Script for Eager version of parallel LU factorization (hands-off)

Notice that there are two very different contexts in which this script may be used. The first is a true shared-memory environment in which the underlying hardware provides the support that would allow for a simple line-by-line translation of this code to function as it should. The other case involves mapping onto a machine whose memory is distributed. While the first case is rather uninteresting from the perspective of the work to be presented here, consideration of the second case brings up a number of issues that merit further examination.

Here again, we have something of a strategy bifurcation. The user may either handle the issues that arise “by hand” or they can suppose that an underlying library, coupled with the script translator, provides the required support. The former option requires a less sophisticated library, a simpler script translator, and seems to hold out the promise of more complete code modularity while the latter would seem to provide a framework for simpler coding. There are a number of issues to be dealt with if the automated code generation system is to work on a distributed machine. In the following sections, we discuss some of these issues and, in the end, construct an LU decomposition algorithm that, while explicitly dealing with the issues involved, does not take on the kind of apparent additional complexity that is traditionally associated with converting an algorithm to a distributed-memory model.

Impact On Abstraction

Let us assume that the SUMMA [73] approach to the implementation of the computational components of these algorithms is the one employed. This approach involves the application of a series of parallel, blocked operations. Using SUMMA, a parallel matrix-multiplication consists of a series of panel-panel (outer-product), matrix-panel (a matrix multiplied by many vectors), or panel-matrix multiplications. The use of SUMMA implies that two other abstractions are required if one does not wish to adopt the “hands-off” stance discussed in the previous section. We refer to these abstractions as *duplication* and *consolidation*.

It may appear difficult to determine whether these operations are more properly referred to as manipulations or computations. However, as we defined computations to encompass any operation that involves more than one *data* object, by definition both abstractions fall into that category. Duplication involves duplicating part of a data object. That is, copying the data from one object into the data component of some other object(s). Consolidation (often referred to as “reduction”) is the converse of this relationship. It involves applying a function (in Fig. 3.9, addition) to some set of objects that may be distributed across the grid and copying the result into another object.

Revisions For Performance

While generating efficient, parallel code from a script is useful, it may be that the code generation system user feels too far removed from the implementation. Sometimes this distance is desired, as in the case of a user who has neither the desire nor the expertise to avail himself of the “deeper” aspects of the programming environment; but often, it is not.

A common mistake that this code generation system avoids is the *permanent* hiding of parallelism and other details. By allowing the user to address the underlying architectural system at different levels of granularity, superb performance and simplicity can be achieved with a reasonably consistent programming “look and feel.” This approach would seem to be the natural extension of the belief that computational abilities (such as parallelism) should not be hidden even though we may wish to conceal how they operate [8, 75].

To illustrate the manner in which such revisions might appear in a script language, we present Figure 3.10. A few remarks about some of the notation used in this “hands on” script are probably called for. The use of the “Local” functional notation is intended to impose the requirement that the enclosed operation does not involve any interprocessor communication. The other two, somewhat cryptic, notations `|*` and `-*` indicate “all processor columns” and “all processor rows,” respectively.

As can be seen in Chapter 5, there are many things that can be determined and used to advantage if the input is more specific than a mathematical description of the problem at hand. In the case where such additional information is withheld from the analysis engine, certain defaults are assumed. However, there is no guarantee that the default values are a good approximation to those of the problem under consideration. It would be very difficult to

```
L === A ; // {Recursive} {Permanent}  (* Establish name equivalence *)
U === A ; // {Recursive} {Permanent}
L has_property unit_lower_triangular ; // (* Permanent Property *)
U has_property      upper_triangular ;
A has_property square ; // (* Actually, Square here *)
partition A =>    / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  where ATL is 0 by 0 ;
do until ABR is 0 by 0
   partition      / ATL # ATR \
                  |###########|
                  \ ABL # ABR /
                                  =>     / A00 # A01 | A02 \
                                         |#################|
                                         | A10 # A11 | A12 |
                                         |-----#-----------|
                                         \ A20 # A21 | A22 /
                  where A11 is local           and
                        A11 is locally square  and
                        A11 is nb by nb ; // No larger than is really implied

   function_override("PLALu1");
   A11 = (L11\U11) <- lu_fact(A11) ;
   Lower[L11tri] |* <-  Lower[L11] ;
   A12 = U12 .<- Lower[L11tri]^-1 * A12 ;
   U11tri -* <-  Upper[U11]  ;
   A21 = L21 .<- A21 * Upper[U11tri]^-1 ;
   L21col |* <-  L21 ;
   U12row -* <-  U12 ;
   A22 .<-  A22 -  L21col * U12row  ;
   partition
                  / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  <=   / A00 | A01 #  A02 \
                                      |------------------|
                                      | A10 | A11 #  A12 |
                                      |##################|
                                      \ A20 | A21 #  A22 / ;
enddo;
```

Figure 3.10: Script for Eager version of **parallel** LU factorization

provide such assurances, as the same implementation must work with different mathematical objects and on different computational grids.

The information that can be communicated via the PLAWright annotations includes:

- The absolute or relative object sizes
- Known constraints or preferences (maximum memory consumed)
- Target architecture or hardware system specifics (per processor or for entire machine)
- Minimum/Maximum/Specific grid size and topology to be used
- That the data will be distributed in some particular manner
- The form of results that are expected from static analysis (see Chapter 5 for options).

3.3 Related Work

Since the work in this chapter considers abstraction in the light of both library construction and programming environment, work related to each topic is discussed.

3.3.1 Library-Based Abstractions

The first issue that should be dealt with is the use of the term *environment* as it applies to a *library*. We posit that a library qualifies as an environment, or "framework" if the reader prefers, because it implicitly imposes a set of concepts on the user. These concepts are expected to be appropriate for the problem at hand and capable as acting as guides for the user.

Libraries are a means to "export" the expertise of some set of people so that it is available to a second set of individuals. Often it is the case that this second set lacks some, or all, of the area-specific expertise of the first group. Most usually the library is considered to be at a "lower-level" than the applications which use it. However, this is not always the case.

Consider the fact that a library can be distributed in at least two forms [57, 54]. The first is the more traditional: computer-language (source or machine) code. The second is in the form of an algorithmic description of the process of concern. This latter form provides an *unrealized* (potentially high-level) functionality set that imposes fewer restrictions, but supplies the same framework as a coded library.

Two well-known examples of traditional linear algebra libraries are LINPACK [22] and LAPACK [5]. Both libraries are built around an index-based scheme combined with a set of general computational kernels. LINPACK, predating LAPACK, utilizes a subset of the kernels exploited by LAPACK. Whereas LINPACK uses only Level-1 BLAS (vector-vector) operations, LAPACK uses all three levels of the BLAS.

While a paper or template [10] library does not provide an application programming interface (API), it does provide, in many cases, a "plan of attack" for implementing a software system and a foundation for creating an API (modulo programming language constraints).

3.3.2 Programming Environments

Two well-known examples of modern programming environments are the Mathematica [77, 35] and MATLAB [58] programming packages. Both supply the user with a vast array of functions for computation and visualization as well as a rudimentary integrated debugging system. Additionally, both provide a huge assortment of library routines and their own programming language with which to call them. Further, both supply interface routines and documented specifications so that the user is allowed to link in routines written in other more traditional languages, such as C or Fortran.

Although Mathematica and MATLAB are examples of environments, they are, in many ways, atypical of such packages, though probably typical of the direction in which these products are moving. While motivations of a commercial nature may keep the source code of these newer systems under wraps for the near future, these products allow the user to plug-in their own modules. [1]

Older software systems tended to be monolithic and, as they did not produce code, plugging in user-defined modules was difficult. Newer packages take a two-tiered approach: those users who wish to continue to view functions as black-boxes are free to do so, while those who want to look inside are given the ability to do so.

3.4 Chapter Summary

In this chapter, we have presented a language that allows the algorithm designer to specialize their operations. Specifically, we have seen that the user is free to manipulate the distribution of data across the computational grid as he sees fit. Such freedom is desirable from a performance-based point-of-view, but it is *necessary* from a flexibility standpoint. If this multi-layered approach is abandoned, the lack of a particular library module may imply that the algorithm designer is engaging in a futile effort. Just as in the sequential case, it is vital that the user have the tools needed to construct novel algorithms.

In Chapter 4 we demonstrate that different script variants result in the production of different code instances, as one would expect. In that same chapter, we describe how this occurs and why it is often beneficial. While Chapter 4 also contains a discussion related to script versions and the differences in the code corresponding to those versions, much of the discussion regarding the importance of this feature is delayed until issues of performance are considered in Chapter 5.

[1]MATLAB supplies, at an added cost, the ability to compile their code into a more efficient executable.

Chapter 4

Automated Code Generation

Implementation tweaking is a standard part of the process when one is developing high-performance scientific applications intended to run on parallel architectures. In this area of research, algorithmic restructuring and code-level optimizations have traditionally been done by different groups [32]. Unfortunately, information that could be employed to make code more efficient is traditionally obscured in the translation from a high-level description into low-level code. Allowing the user to code in a domain-specific language such that high-level information is retained while automatically coupling the requirements to low-level routines would allow for both high- and low-level optimizations. The work presented in this chapter allows one to perform precisely this activity. That is, to generate code instances with high-performance characteristics while programming at a very high level.

For an overview of the automated segment of the process described in this dissertation, the reader is instructed to refer to the illustration in Figure 3.1. There, the high-level program (expressed in PLAWright) is translated into a series of PLAPACK library calls.

The transformation process depends on the specifics of both the target library and the computational environment. Thus, the library routines in the *target* library are annotated with the following in order to create the corresponding *annotated* library:

- Their semantics, which indicate what linear algebra operation is performed (i.e. service provided).
- Guards, which indicate the conditions under which the library call is well-defined.
- Performance characteristics, which are used to generate automated analysis.

The PLANALYZER uses the semantics and guards of the library routines in order to generate a number of implementations whose functionality corresponds to the input script version. This process is the focus of this chapter as is indicated in Figure 4.1. While this chapter largely ignores performance considerations, the next chapter focuses on the issue of performance characteristics, so the reader with such concerns need not worry that they have been entirely overlooked.

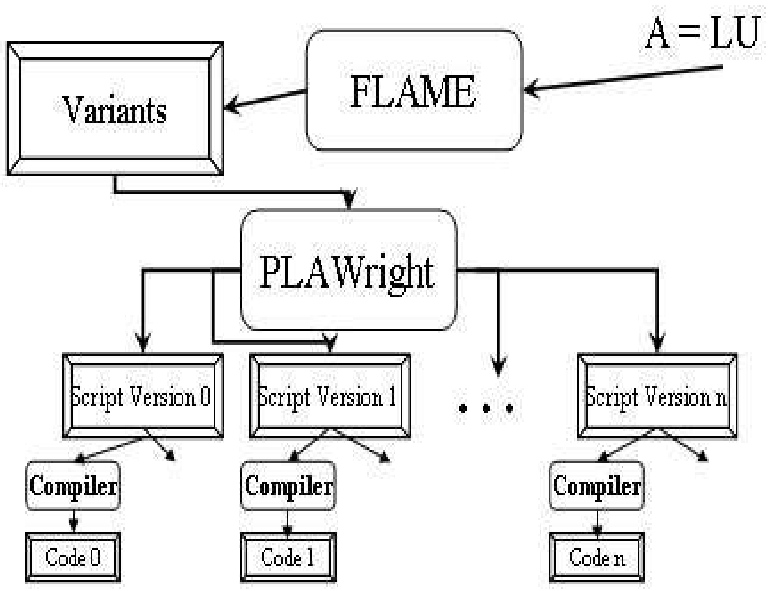


Figure 4.1: Where the code generator fits into the "grand scheme" of things.

4.1 Motivation for Automating Library Linkage

There are many reasons that one might wish to automate library linkage. In Chapter 3, the PLAWright script language was presented. In that chapter, the focus was on the fact that the language provided for the efficient utilization of the expertise of the programmer. It was also pointed out that the scripts could be compiled and the resulting codes were computationally efficient. By automating library linkage, one can write a single exemplar code (script) that compiles into many different code realizations.

This practice also facilitates the leveraging of the expert's knowledge via a separation of concerns. The application writer can concentrate on the picture as he sees it and rely on the fact that the library writer provides efficient routines *and* that those routines are linked to at the time of compilation. The library writer might have a similar relationship with the kernel writer. All of these users could be "communicating" their work through the annotations they add to their contributed routines and allowing the compilation system to find a match between what they require (as is expressed in the script) and what the library provides (as is communicated in the associated annotation). Thus, the automated system represents an potential extension to what is often-sought in this relationship among programmers. In the next chapter, we discuss how high performance is achieved. Here, we assume that efficient routines are linked to the user's requests.

Portability can be as important as performance in the domain of dense linear algebra libraries. Not only do companies come and go, but vastly different architectural designs may be created and tested. Sometimes this testing occurs in the marketplace and sometimes it transpires in research facilities, but the shakeout that determines what lasts and what does not, will continue to happen as long as resources are finite. The core difficulty here is how to design a code generation system or systematize an approach such that the result is amenable to both evolutionary (e.g. Cray T3D $\rightarrow$ Cray T3E) and revolutionary (e.g. Intel Paragon $\rightarrow$ LEGION $\rightarrow$ Blue Gene) changes.

It would seem that adapting to changes that are, as measured by performance metrics, orders of magnitude apart would best be supported by two distinct approaches [3], one emphasizing ease-of-use, and the other concentrating solely on achieved performance. However, it is our thesis that one should moderate, at times, the (laudable) goal of "a separation of concerns." One must determine when concerns are identical or largely overlapping (i.e., to decide if and when these concerns are the same when viewed from a given level of abstraction).

The scripts corresponding to the Eager and Lazy versions of LU factorization (depicted in Figures 4.2 and 4.3, respectively) are in a form that might be termed user-friendly. However, the user may wish to give directives to the code generation system. These directives might involve object distribution, block sizes, or specifying the name of a specific library routine. In this chapter we address the impact of these "hints" on code production. For example, if one were to specialize Figure 4.2 by providing such hints, the result might well be Figure 4.4 (seen previously in Figure 3.10). Note that lines 22 and 24-30 in Figure 4.4 are all user-supplied hints related to function selection (24) or distribution specification (24-30).

4.2 Issues in Library Linkage

The issues that one must consider when designing, in the abstract, an automated library-linkage system are mirrored when one's focus shifts to an implementation. This section restricts itself to issues that apply to the abstract case while the next section deals with each issue in the context of a proof-of-concept implementation. The manner in which the process of linking takes place is delayed until Section 4.3 because communication regarding that subject benefits from the existence of concrete examples.

4.2.1 A (Fictitious) Linking Library

There are many ways in which any scientific software library can be constructed. We restrict our attention to theoretical constructs that lie at opposite ends of the spectrum of possibilities and consider issues germane to the use of a script language such as PLAWright.

First, there is the possibility that the library contains a great many subroutines. So many, in fact, that there is always at least one subroutine that matches the feature requirements of any operation requested (see Section 4.2.2) by a script statement. At the other end of the scale, there is the possibility that the library consists of few routines, but

```
L has_property unit_lower_triangular ; // (* Permanent Property *)
U has_property      upper_triangular ;
A has_property square ; // (* Actually, Square here *)
L === A ; // {Recursive} {Permanent}  (* Establish name equivalence *)
U === A ; // {Recursive} {Permanent}
partition A =>    / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  where ATL is 0 by 0 ;
do until ABR is 0 by 0
   partition      / ATL # ATR \
                  |###########|
                  \ ABL # ABR /
                                  =>     / A00 # A01 | A02 \
                                         |#################|
                                         | A10 # A11 | A12 |
                                         |-----#-----------|
                                         \ A20 # A21 | A22 /
                  where A11 is local           and
                        A11 is locally square  and
                        A11 is nb by nb ; // No larger than is implied

   A11 = (L11\U11) <- lu_fact(A11)  ;
   A12 = U12 <- L11^-1 * A12 ;
   A21 = L21 <- A21 * U11^-1 ;
   A22 <-  A22 -  L21 * U12  ;
   partition
                  / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  <=      / A00 | A01 #  A02 \
                                         |------------------|
                                         | A10 | A11 #  A12 |
                                         |##################|
                                         \ A20 | A21 #  A22 / ;
enddo;
L =!= A;
U =!= A;
```

Figure 4.2: Computer-readable Script for Eager version of LU factorization

```
L has_property unit_lower_triangular ; // (* Permanent Property *)
U has_property upper_triangular ;      // (* Same as non-unit *)
A has_property square ; // (* Actually, Square here *)
L === A ; // {Recursive} {Permanent}  (* Establish name equivalence *)
U === A ; // {Recursive} {Permanent}
partition A =>    / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  where ATL is 0 by 0 ;
do until ABR is 0 by 0
   partition      / ATL # ATR \
                  |###########|
                  \ ABL # ABR /
                                 =>    / A00 # A01 | A02 \
                                       |#################|
                                       | A10 # A11 | A12 |
                                       |-----#-----------|
                                       \ A20 # A21 | A22 /
                  where A11 is local           and
                        A11 is locally square  and
                        A11 is nb by nb ; // No larger than is implied
  A01 = U01 <- L00^-1 * A01 ;
  A10 = L10 <- A10 * U00^-1 ;
  A11 = (L11\U11) <- A11 - L10 * U01 ;
  A11 = (L11\U11) <- lu_fact(A11)  ;
  partition
                  / ATL # ATR \
                  |###########|
                  \ ABL # ABR /   <=     / A00 | A01 #  A02 \
                                       |------------------|
                                       | A10 | A11 #  A12 |
                                       |##################|
                                       \ A20 | A21 #  A22 / ;
enddo;
L =!= A;
U =!= A;
```

Figure 4.3: Computer-readable script for Lazy version of LU factorization

```
L === A ; // {Recursive} {Permanent}  (* Establish name equivalence *)
U === A ; // {Recursive} {Permanent}
L has_property unit_lower_triangular ; // (* Permanent Property *)
U has_property      upper_triangular ;
A has_property square ; // (* Actually, Square here *)
partition A =>    / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  where ATL is 0 by 0 ;
do until ABR is 0 by 0
   partition      / ATL # ATR \
                  |###########|
                  \ ABL # ABR /
                                  =>     / A00 # A01 | A02 \
                                         |#################|
                                         | A10 # A11 | A12 |
                                         |-----#-----------|
                                         \ A20 # A21 | A22 /
                  where A11 is local           and
                        A11 is locally square  and
                        A11 is nb by nb ; // No larger than is really implied

   function_override("PLALu1");
   A11 = (L11\U11) <- lu_fact(A11) ;
   Lower[L11tri] |* <-  Lower[L11] ;
   A12 = U12 .<- Lower[L11tri]^-1 * A12 ;
   U11tri -* <-  Upper[U11]  ;
   A21 = L21 .<- A21 * Upper[U11tri]^-1 ;
   L21col |* <-  L21 ;
   U12row -* <-  U12 ;
   A22 .<-  A22 -  L21col * U12row   ;
   partition
                  / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  <=   / A00 | A01 #  A02 \
                                      |------------------|
                                      | A10 | A11 #  A12 |
                                      |##################|
                                      \ A20 | A21 #  A22 / ;
enddo;
```

Figure 4.4: Annotated script for an Eager version of parallel LU factorization

routines from which one could construct an algorithm matching the semantic requirements of any legal script statement.

Either of these libraries can be used in an automated code generation system. Determining which one is "best" would seem to be a philosophical, not scientific, issue. Certainly, in the large library case, matching the requirements of the script to the functionality provided by the library is simpler. The matching can be both 1:1 on a line-by-line basis and purely syntactic in the first case. Further, if the underlying library is optimized, the operations corresponding to these matches is almost always the best choices from a performance perspective. In the small library case, the matching procedure is more complex, as it has the responsibility of building programs from components.

For the purposes of this dissertation, we focus on a library that lies somewhere in the middle. This is justified for the following reasons. First, if the large, efficient library is considered the target, the work involved in the binding process is not very interesting. In that case, matching is simple and, while automated performance analysis (see Chapter 5) might be interesting, it is not necessary, as the highest degree of available efficiency is virtually assured simply by dint of the "brains" in (or behind) the library. Second, the case of the building-block library has an unfortunate stopping point, namely the constructs in the language of output. Since the idea of generating optimized assembly language from a high-level script language would appear to be too ambitious for any single dissertation, a middle ground was selected. In any case, expertise is required. For the large library, a great deal of expertise would be needed to construct the annotations, while in the building-block library case, the greater expertise would be required to transform the input to a list of library-matchable requirements. Finally, the PLAPACK library was targeted because it is an implementation of the layered approach advocated in this document and has good performance characteristics.

4.2.2 Reducing a Script

The algorithm expressed in script form is to be realized through the functionality of a library, thus the requirements of the script must be mated to that library. One could match the requirements directly, if they were to assume the "large" version of the library described in Section 4.2.1. However, that section clarifies why the use of such a library is not employed in this work. Thus, we assume that some form of reduction to requirements must take place.

The question then becomes one of determining the language into which these requirements are translated. This determination has been largely dictated to us by the abstractions behind the language itself. In Section 3.2.1, details about the necessary abstractions underlying the PLAWright language were given. It would seem certain that the language form we employ to express the script requirements must have the ability to express those abstractions. Certainly, though it is not strictly necessary, it can also prove beneficial if this "down-translation" (from script to requirements) is capable of producing script-induced-requirements that express higher-level needs. It is often advantageous to stay as close as possible to the application (and the application *language*) so as not to lose information. Therefore, we deem it beneficial for any such code generator to have the ability to translate down to various levels of feature abstractions so that it can match the library at the highest level possible. Alternatively, translation could occur in a step-wise fashion, where library functionalities are matched at the highest level available and further refinement (down-translation) performed on a need-driven basis. As can be seen in the implementational arena (Section 4.3.3), the former approach was selected strictly for reasons of expediency.

4.2.3 Annotating a Library

It might appear that the questions regarding the form of the language used to annotate the library have already been answered. Section 4.2.2 supplied details about how the abstract down-translation is to occur, and it seems logical to assume that the library annotations are to match that language if a binding is to occur. Unsurprisingly, here, we do make that assumption. Surprisingly, this is not the end of the subsection.

It would seem that we are still left with some choices about the language we wish to use in order to annotate our fictitious library. We could:

1. Use the target language of the script requirements (lowest level).
2. Employ PLAWright to annotate the library and the script translation engine to "digest" those annotations.
3. Exploit a combination of the first two ideas.

We utilize the third option. However, for purposes of exposition, a mid-level format is used to illustrate the realization of these annotations.

4.2.4 Producing Output

The kind of output produced has largely been determined by the methodological approach we have assumed: the use of some existing library or libraries. Since interoperability concerns are outside the scope of the research completed, we have restricted ourselves to a single computer language. Further, because the existing scientific libraries are usually written in an imperative language, most often C or Fortran, we restrict our attention to those languages.

4.3 Implementation: An Automated Library

The software system depends on matching script requirements to the library functionality. Thus, it avoids having to handle many of the difficulties involved when one deals with novel architectures by relying on a library expert. This expert is expected to provide the (PLANALYZER) system with correct (functionality and performance) annotations. Further, it is expected that the routines to be "mined" evince superb performance characteristics.

Those disclaimers aside, not all is lost. In the discussion of Section 4.2.1 regarding the design of a fictitious library, it was pointed out that the code generator can compose a fairly small number of primitive operations to implement an algorithm. This removes a good deal of the burden from the shoulders of the library expert as that individual can be supplied with a short list of annotated and optimized functions which must be provided. While it is still true that the expert may have to do some work for this to be achieved, the burden is decidedly eased when compared to traditional library building methods. In those cases, supplying the kernel routines was the first of many steps; here, it marks the shift into a far more automatic method of development.

4.3.1 Tools Employed

In order to allow automated binding to an annotated library, a number of software tools were used. The first step in the chain of execution is the ANTLR [61, 62] compiler-compiler. Given PLAWright code, ANTLR was used to compile the scripted input into a functional programming form that was syntactically well-formed *Mathematica* input code. At that point in the process, *Mathematica* [77] is utilized in order to perform the pattern-matching necessary to combine the requirements of the program with the functionality provided by the (annotated) library, and to translate this intermediate form into an executable largely composed of calls to the target library.

4.3.2 PLAPACK: A Target Library

When coupling a script to a library, it is beneficial for the library to be constructed in accordance with the same design philosophy reflected in the script language. PLAPACK is well-suited to this goal, due to its layered structure. Figure 4.5 illustrates the PLAPACK library's layered nature and meshes nicely with this design goal.

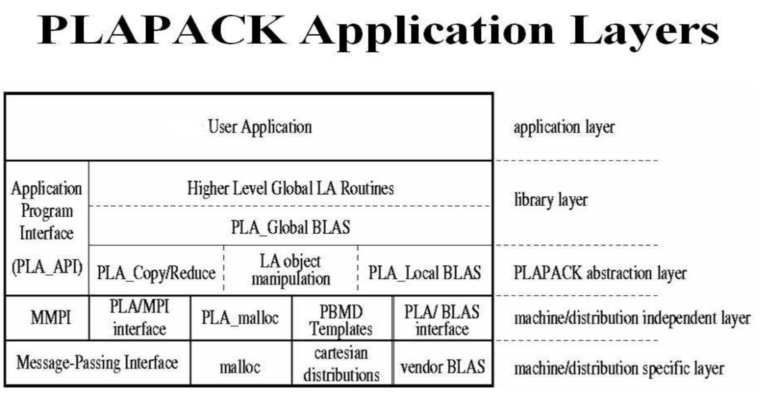

PLAPACK Application Layers
User Application
application layer
Application Program Interface (PLA_API)
Higher Level Global LA Routines
PLA_Global BLAS
library layer
PLA_Copy/Reduce
LA object manipulation
PLA_Local BLAS
PLAPACK abstraction layer
MMPI
PLA/MPI interface
PLA_malloc
PBMD Templates
PLA/ BLAS interface
machine/distribution independent layer
Message-Passing Interface
malloc
cartesian distributions
vendor BLAS
machine/distribution specific layer

Figure 4.5: The layered structure of the PLAPACK library

Very briefly, the layering allows the naive user to program at a very high level, so as to interact strictly with high-level global routines and the shared-memory view afforded by the use of the (poorly named) "API" routines. The more expert user may exercise greater control of the process by utilizing the lower levels of the library. This allows the application programmer to create a working proof-of-concept algorithm, and then to iteratively refine it in order to maximize performance [2]. The work presented here further eases this process by automating optimizations and allowing the user to program at an even higher level of abstraction if he so chooses and to spend more of their energy on algorithmic, rather than programming, refinement.

4.3.3 Compiling PLAWright

The compilation of a PLAWright script is most easily thought of in terms of rewrite rules, syntax-based tranformations. One form of the implementation uses a simple table of rewrite rules in order to perform this translation. As that is an approach that lends itself to exposition, that implementation is the one that is studied in this section.

Consider line 25 in Figure 4.2.

```
A22 <- A22 - L21 * U12 ;
```

After the stage of compilation handled by the ANTLR compiler tool has been performed, the intermediate form of the program is in a format that can be parsed by *Mathematica*. The ANTLR tool also determines if the script is syntactically correct, but the *Mathematica* engine is responsible for determining whether or not the script can be transformed into an executable program and, if so, how.

When this line of code enters *Mathematica* it has the following form:

```
AssignTo[ A22, PLAMinus [ A22, PLATimes[ L21, U12 ]]]
```

which is transformed, by default, into:

```
AssignTo[ A22, PLAPlus [ A22, PLATimes[ -1, L21, U12]]]
```

The code generator explores many paths of translation. Let us consider one of the eventual targets of this translation:

```
PLAGemm[transa_, transb_, alpha_, A_, B_, beta_, C_]
```

We can ignore the `transx_` parameters, as the details might prove distracting. In order to arrive at this format, the initial form must be transformed into one that matches the `PLAGemm[]` call. The following line illustrates the format that must be matched (the checks of object types that are included in the rewriter are omitted for brevity). The following line is intended to capture the features of the `PLA_Gemm( )` library function, but the description is divorced from that particular implementation.

```
AssignTo[C1_,PLAPlus[PLATimes[alpha_, A_, B_],
                     PLATimes[beta_, C2_ ]]]
```

A few topics need to be dealt with here. The first involves the fact that `C1_` and `C2_` both match `A22`. This is allowable in unification as two variables can match the same object. The second requires only slightly more explanation. Barring explicit user directives to the contrary, the rewriting system can change the order of the objects involved in an addition operation. Therefore,

```
PLAPlus[ A22, PLATimes[ -1, L21, U12]]
```

becomes

```
PLAPlus[ PLATimes[ -1, L21, U12], A22]
```

in one search chain. The third and final issue involves the multiplication by scalars. The operation to be matched includes `alpha_` and `beta_` terms that are not in the original operation. This can be handled in at least two ways. One solution is to default values to the operations (in the case that no scalar is supplied). Alternatively, one could build knowledge into the rewriter (e.g., that multiplication by 1 results in an object with unchanged values). The second option was utilized in the engine for reasons of expediency, but this will likely be changed in the future, as dealing with such things using a demand-driven approach tends to be more computationally efficient.

Given that the PLANALYZER eventually matches:

```
PLAGemm[transa_, transb_, alpha_, A_, B_, beta_, C_]
```

all that is left is the output of code. This is a simple step involving a simple `Expression[]` to `String[]` rewrite inside *Mathematica* resulting in the output:

```
PLA_Gemm(PLA_NO_TRANS, PLA_NO_TRANS, mscalarnegone, L21, U12, mscalarone, A22);
```

4.3.4 Annotating the Library: Functionality Provided

To apply any operation, the preconditions of that operation must be met in order for the semantics of the operation to be well-defined. Therefore, tests are applied in order to determine if the function is applicable to the "current state" of the program, as seen through the eyes of the code-generation mechanism. In order to advance the state of the program, the applicable and required operations are applied to the current state.

Pre-Conditions: Guards

Consider a simple example consisting of the following one-line high-level program.

$$\texttt{A} \leftarrow \texttt{A} \times \texttt{C;}$$

The PLANALYZER attempts to match this with the PLAPACK library's functionality and after some analysis identifies the following call as a possible match.

```
AssignTo[A, PLAPlus[PLATimes[mscalarone, A, C],
                  PLATimes [mscalarzero, A]]]
```

where `mscalarone` and `mscalarzero` correspond to 1 and 0, respectively.
The above is an instance of the library call

```
AssignTo[C_, PLAPlus[PLATimes[a_, A_, B_], PLATimes[b_, C_]]]
```

whose general functionality is

$$\texttt{C\_} \leftarrow (\texttt{a\_} \cdot \texttt{A\_} \times \texttt{B\_}) + (\texttt{b\_} \cdot \texttt{C\_})$$

where `a_` and `b_` are unifiable variables that can be thought of as being of type scalar and `A_`, `B_`, and `C_` are unifiable variables of type matrix (with conformal dimensions). The guards specify that neither `A_` nor `B_` can be the same as `C_`, therefore,

```
AssignTo[A, PLAPlus[PLATimes[mscalarone, A, C],
                  PLATimes [mscalarzero, A]]]
```

is not a valid transformation. Thus, a new variable, used to hold a copy of `A`, is declared. This allows the use of a `PLA_Gemm( )` call while satisfying the guards.

This creates the following chain of operations:

```
PLA_Matrix_create_conf_to(A, &MATRIXTEMPA123);}
PLA_Copy(A, MATRIXTEMPA123);}
PLA_Gemm(PLA_NOTRANS, PLA_NOTRANS, mscalarone, A, B,
                                   mscalarzero, MATRIXTEMPA123);
PLA_Copy(MATRIXTEMPA123, A);
```

For reasons detailed in Chapter 5, this code will be rejected due to its inherent inefficiencies, but it is *one of* the paths that will be explored.

Post-Conditions: Adds and Deletes

To advance the state of the computation, the operations are applied to the current state. For the purposes of code generation, application means being added to the program under construction; in the context of state advancement, it means having the appropriate properties added to and deleted from the property set that corresponds to program state.

A simple example should clarify this procedure. Reconsider the aforementioned "chain" of code.

```
1.  PLA_Matrix_create_conf_to(A, &MATRIXTEMPA123);}
2.  PLA_Copy(A, MATRIXTEMPA123);}
3.  PLA_Gemm(PLA_NOTRANS, PLA_NOTRANS, mscalarone, A, B,
                                      mscalarzero, MATRIXTEMPA123);
4.  PLA_Copy(MATRIXTEMPA123, A);
```

At the outset (the non-existent line 0), matrix `A` had some state (size, shape, etc.) while `MATRIXTEMPA123` had no such properties. After the execution of lines 1 and 2, `MATRIXTEMPA123` has the same properties as `A` and could be used as a substitute for `A`. However, after the "execution" of line 3, `MATRIXTEMPA123` has had some of those properties deleted (e.g. that its data component is the same as `A`'s), has had some left unchanged (e.g. the size of the matrix), and has had some added (e.g. that its data component is the product corresponding the matrix-multiplication). After the execution of line 4, the properties of the two objects again coincide.

4.3.5 Producing Output

The script language must be translated into a compilable language. The viable alternative would be to have the translation system transform the input down to the level of assembly code, but that part of optimizing-compiler technology is not part of this dissertation (as was alluded to in Section 4.2.1). Therefore, the target language is an issue that must be considered in the realization of the code generator.

First, we must consider which programming language(s) we wish to target. Many issues arise in such a decision. Since FLAMBE has been written in both Fortran and C, we target a parallel version of FLAME, PFLAMBE. [1]

The translation of the algorithm into efficient code has clearly defined lines of demarcation. This design decision allows language independence for as long as is possible in the compilation process. The stratification of the FLAME $\rightarrow$ PLAWright $\rightarrow$ PLANALYZER system is such that new programming languages might be targeted in the future.

4.3.6 A Realized Construction

When the PLANALYZER was supplied with the script depicted in Figure 4.2, it produced many different coding instantiations. One of these is depicted in Figure 4.6

While the generated library routines shared many traits, they did evince some differences. The most common of these was the creation of temporary objects for the storage of matrices that would act as temporary copies for the computations performed. In the case of Eager LU factorization, this seems rather illogical, but, it is not universally so. For example, if the following computations were to occur:

[1] PFLAMBE is a sugarcoated extension of the PLAPACK language expressed in the FLAMBE manner. PFLAMBE was selected to be the target language because its format is not in flux. In addition the use of PFLAMBE allows us to study more deeply nested memory hierarchy issues in Chapter 5.

```
for(;;)
    {
    PLA_Obj_global_length( ABR, &PLAEnderLength);
    PLA_Obj_global_width( ABR, &PLAEnderWidth);
    if( PLAEnderLength == 0 && PLAEnderWidth == 0) break;
    PLA_Obj_split_size( ABR , PLATOP , &PLAlength2, &dummyint );
    PLA_Obj_split_size( ABR , PLALEFT , &PLAwidth2, &dummyint );
    nb = min (PLAlength2 , PLAwidth2 );
    PLA_Obj_view_all (ATL,  &A00);
    PLA_Obj_vert_split_2( ATR, nb ,  &A01,  &A02 );
    PLA_Obj_horz_split_2( ABL, nb ,  &A10,
                                     &A20 );
    PLA_Obj_split_4( ABR, nb, nb ,  &A11,  &A12,
                                    &A21,  &A22 );
    PLA_Local_LU(A11);
    PLA_Trsm( PLA_SIDE_LEFT , PLA_LOWER_TRIANGULAR , PLA_NOTRANSPOSE ,
              PLA_UNIT_DIAG , mscalarspecialone , A11 , A12 );
    PLA_Trsm( PLA_SIDE_RIGHT ,PLA_UPPER_TRIANGULAR , PLA_NO_TRANSPOSE ,
              PLA_NONUNIT_DIAG , mscalarspecialone , A11 , A21 );
    PLA_Gemm( PLA_NO_TRANSPOSE , PLA_NO_TRANSPOSE ,
              mscalarspecialnegone , A21 , A12 , mscalarspecialone , A22 );
    PLA_Obj_join_4( A00, A01,
                    A10, A11,  &ATL );
    PLA_Obj_horz_join_2( A02,
                         A12,  &ATR );
    PLA_Obj_vert_join_2( A20, A21,  &ABL );
    PLA_Obj_view_all( A22,  &ABR );
    }

```

Figure 4.6: Central loop of created code for the Eager variant of LU factorization

```
A <- B * C;
A <- E;
D <- B * C * B * C;
```

it might make sense to create shadow storage for the `B * C` result. In any event, the same compiler technology that is used to determine how to allocate registers most efficiently can be employed here for entire matrices.

Later, in Chapter 5, we revisit why such differences exist among the produced coding instances and what they lend the system as a whole.

4.3.7 Libraries

We focus on two libraries that have very similar functionality for the purposes of the research presented here.

ScaLAPACK

The ScaLAPACK library is a parallel extension of the LAPACK library designed for maximal code re-use. The goal of the ScaLAPACK project is to implement all of the LAPACK routines in an efficient manner on a variety of parallel architectures. Through code reuse (of the LAPACK library), the project attempts to use existing optimized and tested serial code on each processor of a parallel machine. This is done through an intermediate level called the PB-BLAS (Parallel Blocked BLAS) [14] in an attempt to supply users with a layered-library.

Unfortunately, it is our opinion that ScaLAPACK to sacrifices *some* design coherence, or at least readability, in order to gain this code-leverage. This is not surprising as the character of the software is heavily influenced by the bottom-up nature of this approach. Higher-level parallel routines may call lower-level parallel (or serial) routines that do not share the same design goals. This may result in unfortunate communication penalties. Further, the parallel versions of serial subroutines tend to have many additional parameters, due to the increased indexing complexity. This tends to make these routines somewhat difficult to use and the underlying library somewhat difficult to maintain.

In addition, ScaLAPACK ties itself to the BLACS communications library. While the coupling of two libraries may or may not be a problem, there appear to be some problems with the BLACS in that there are simple global communications patterns that it appears to lack.

PLAPACK

Like ScaLAPACK, PLAPACK [74] is a library that can be used for doing dense linear algebra on parallel computers. PLAPACK differs from ScaLAPACK in that it is an object-based construct that insulates the user from error-prone index computations through the use of "views." Views allow objects to co-reference portions of the same data (e.g. parent objects may hold data that can be manipulated by any number of children).

4.3.8 Library Binding

A claim is sometimes made that no *class*[2] of user wishes to view the libraries that they utilize as black box routines. This stands in contrast to the fact that the typical user of a package such as MATLAB is assumed not to care about what is underneath. In truth, it is often the case that users do not wish to *have* to know what is going on underneath, but want the *option* of ascertaining and leveraging such knowledge. Projects such as FALCON [20, 57, 19] have been very successful in automatically restructuring MATLAB code into languages such as Sage++ and Fortran90. More recent efforts such as Broadway [50] have made strides towards allowing the user to produce high-performance code while programming in a somewhat naive manner. This is facilitated by a sophisticated, optimizing compilation system. This obviates the need for expertise to some degree, but allows for the leveraging of programmer-originated optimizations.

It is important to note the synergistic role between library and compiler in these cases. FALCON utilizes little information about the relationships between routines in the libraries that it uses. Conversely, Broadway exploits such information and benefits from the layered construction of libraries PLAPACK.

4.4 Experimental Results

The PLANALYZER is a proof-of-concept implementation. In Section 2.8, a number of empirical tests were performed with FLAME as the methodology under study. In this section, I demonstrate the efficacy of the PLANALYZER as regards code generation by applying it to a number of algorithmic variants and versions. These algorithms exhibit differing levels of complexity and the resulting codes evince different performance characteristics.

In this dissertation, the concepts underlying an automated system that could be used to generate computer code and analysis for linear algebra algorithms have been discussed. Viewing the components in the context of the automated system as a whole yields an image akin to the one seen in Figure 4.7.

4.4.1 Generating Parallel LU Factorization

In order to create a hybridized algorithm, one must first generate a number of variants of the algorithm under consideration. When using the PLANALYZER, the next step involves translating the algorithms into an input format acceptable to the PLAWright compiler. It is at this time that these scripts are annotated with performance and analytical directives if these specializations are desired. Finally, the scripts are coupled with the annotated library

[2]It may be the case that some *individual* users do wish to do so.

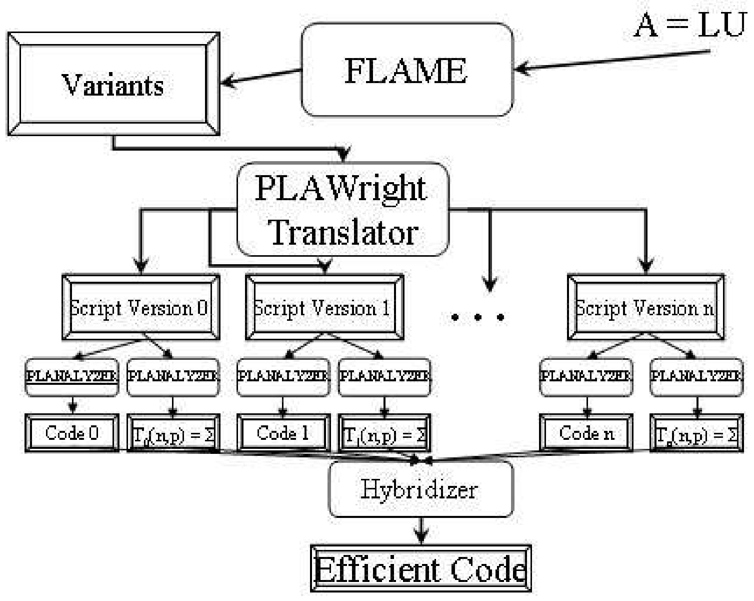


Figure 4.7: The "grand scheme" of things as has been discussed.

in order to generate code and corresponding performance analysis. This section covers these steps and analyzes the results.

Generating the Algorithms

Using the FLAME methodology (see Chapter 2), five common variants of LU factorization were systematically generated as is detailed in Section 2.4. Because the Eager variant tended to yield the best performance for large problems executed on parallel machines, it was selected for specialization in the remainder of the experiments concerning differences between algorithmic versions.

Generating the Scripts

As is discussed in Chapter 3, the barrier between FLAME and the PLANALYZER is bridged by converting the algorithm into an ASCII representation. The differences between the way in which we might depict an algorithm in a technical report and this ASCII version were examined in Section 3.2.1. Some of these scripts were specialized for the parallel environment that was to be the target architecture (PLAPACK v3.1 executing on a Cray T3E). The methods employed to perform this specialization were described in Section 3.1.3.

The Scripts: Details

Let us briefly describe the codes that were analyzed by the PLANALYZER system. First, there were the five algorithmic variants of LU factorization. A corresponding version, in terms of complexity, of each variant was used for both the code generation and analysis tests. The common thread between these variants has to do with the sub-problem of LU factorization. In each case, the submatrix to be factored was localized (via explicit script directives) so as to exist on one processor. No further directives were supplied. The variants tested were:

1. Eager LU Factorization
2. Lazy LU Factorization
3. Row-Lazy LU Factorization
4. Column-Lazy LU Factorization
5. Row-Column-Lazy LU Factorization

In order to further explore the capabilities of the analysis engine, the Eager variant was specialized through both annotation and direct manipulation of a form of the code that would not be available to the casual user. The versions studied were:

1. **Eager1:** The script was specialized to enforce a 1 by 1 blocking. The intermediate code was hand-massages in order to avoid the call to the LU factorization of the 1 by 1 block (avoiding a function call that would result in a NO-OP).

2. **Eager2:** The script was specialized to enforce a 1 by 1 blocking as well as explicitly creating a duplicated-everywhere object (a *multiscalar*) to hold the portion to be factored. Annotations were also added so that the would call local PFLAMBE routines for the triangular solves. The intermediate code was hand-massages in order to avoid the call to the LU factorization of the 1 by 1 block as well as the triangular solve involving a unit-diagonal 1 by 1 matrix.

3. **Eager3a:** Annotations to the script forced the LU-factorization subproblem (A11), to exist on a single processor. This resulted in an LU subproblem of the distribution blocking size. Further, function override was used to force the **Eager1** algorithm (above) to be utilized for factoring the LU subproblem.
   Eager3b: Annotations to the script forced the LU-factorization subproblem (A11), to exist on a single processor. Further, function override was used to force the **Eager2** algorithm (above) to be utilized for factoring the LU subproblem.

4. **Eager4:** Identical to **Eager3a/3b** except that functional override was used to force a call to a handwritten local LU kernel whose performance was assumed to be that of a "standard" level-2 BLAS routine (about 10% of processor peak) when solving the LU decomposition subproblem.

5. **Eager5:** The same as **Eager4** save for the fact that the script was annotated to force a duplication of the object to be factored (a copy into a duplicated-everywhere multiscalar). This allowed the application of local triangular solves, so the script was annotated to enforce that optimization (via the use of `.<-`, instead of `<-`, assignment directives).

6. **Eager6a:** Partitions the matrix to be factored into sub-blocks that are of the algorithmic blocking size (64) rather than the distribution blocking size (16). Functional override was employed in order to call **Eager4** for the LU subproblem. All other operations were global.
   Eager6b: Identical to **Eager6a**, save for the fact that the LU subproblem was handled by **Eager5**.
   Eager6c: Identical to **Eager6a**, save for the fact that the LU subproblem was handled by **Eager1**.

Generating Code

The script variants were, in nature, similar to the one depicted in Figure 4.8. Each of the examined variants was given the same level of annotated direction (see Section 4.4.1) to produce the versions examined.

The codes produced resembled the program in Figure 4.9. For purposes of presentation, comment bars were placed around the section of code that makes this a Lazy algorithm, the name was changed from the unique name generated by *Mathematica* to `Lazy` and the lines containing variable declarations and object "free"s were abbreviated.

A number of code instantiations were produced from each scripted variant input. The number of instantiations could prove misleading so the reader should bear in mind that the number is the product of the number of instantiations available for each line of the script involving an operation and, more importantly, that most of the codes generated were suboptimal. The reason for this latter occurrence is detailed in Section 4.3.6 and is a property of the prototype nature of the PLANALYZER system. The code generation engine and the analysis engine were not employed in concert.

The number of code instantiations produced:

1. Eager LU Factorization: 84

2. Lazy LU Factorization: 84

3. Row-Lazy LU Factorization: 588

4. Column-Lazy LU Factorization: 588

5. Row-Column-Lazy LU Factorization: 5292

While only random samples of the generated codes were examined, the more efficient codes tended to correspond to those that have been generated by hand.

```
L has_property unit_lower_triangular ;
U has_property upper_triangular ;
A has_property square ; // (* Actually, Square here *)
L === A ; // {Recursive} {Permanent}  (* Establish name equivalence *)
U === A ; // {Recursive} {Permanent}
partition A =>    / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  where ATL is 0 by 0 ;

do until ABR is 0 by 0
   partition       / ATL # ATR \
                   |###########|
                   \ ABL # ABR /
                                   =>     / A00 # A01 | A02 \
                                          |#################|
                                          | A10 # A11 | A12 |
                                          |-----#-----------|
                                          \ A20 # A21 | A22 /
                  where A11 is local           and
                        A11 is locally square  and
                        A11 is nb by nb ; // No larger than this

  A01 = U01 <- L00^-1 * A01 ;
  A10 = L10 <- A10 * U00^-1 ;
  A11 = (L11\U11) <- A11 - L10 * U01 ;
  A11 = (L11\U11) <- lu_fact(A11)  ;

partition
                  / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  <=   / A00 | A01 #  A02 \
                                      |------------------|
                                      | A10 | A11 #  A12 |
                                      |##################|
                                      \ A20 | A21 #  A22 / ;
enddo;
L =!= A;
U =!= A;
```

Figure 4.8: PLAWright-compilable script for a Lazy version of LU factorization

```
#include "mpi.h";
#include "PLA.h"
int Lazy(PLA_Obj A)
{
    <variables are declared>
    PLA_Obj_template(A, &MyTemplate);
    /*Create usual constants*/
    PLA_Create_constants_conf_to(A,&mscalarspecialnegone,&mscalarspecialzero,&mscalarspecialone);
    /*UserWarning: Square ShapeSpec not yet enforced ... rule not fired*/
    PLAlength1 = 0 ;
    PLAwidth1 = 0 ;
    PLA_Obj_split_4( A, PLAlength1, PLAwidth1 ,  &ATL,  &ATR,  &ABL,  &ABR );
    for(;;) {
        PLA_Obj_global_length( ABR, &PLAEnderLength);
        PLA_Obj_global_width( ABR, &PLAEnderWidth);
        if( PLAEnderLength == 0 && PLAEnderWidth == 0) break;
        PLA_Obj_split_size( ABR , PLA_SIDE_TOP , &PLAlength2, &dummyint );
        PLA_Obj_split_size( ABR , PLA_SIDE_LEFT , &PLAwidth2, &dummyint );
        nb = min (PLAlength2 , PLAwidth2 );
        PLA_Obj_view_all (ATL,  &A00);
        PLA_Obj_vert_split_2( ATR, nb ,  &A01,  &A02 );
        PLA_Obj_horz_split_2( ABL, nb ,  &A10,  &A20 );
        PLA_Obj_split_4( ABR, nb, nb ,  &A11,  &A12,  &A21,  &A22 );
        /******************************************************************/
        PLA_Trsm( PLA_SIDE_RIGHT , PLA_UPPER_TRIANGULAR , PLA_NO_TRANSPOSE ,
                  PLA_NONUNIT_DIAG , mscalarspecialone , A00 , A10 );
        PLA_Trsm( PLA_SIDE_LEFT , PLA_LOWER_TRIANGULAR , PLA_NO_TRANSPOSE ,
                  PLA_UNIT_DIAG , mscalarspecialone , A00 , A01 );
        PLA_Gemm( PLA_NO_TRANSPOSE , PLA_NO_TRANSPOSE , mscalarspecialnegone ,
                  A10 , A01 , mscalarspecialone , A11 );
        PLA_Local_LU(A11);
        /*******************************************************************/
        PLA_Obj_join_4( A00, A01, A10, A11,  &ATL );
        PLA_Obj_horz_join_2( A02, A12,  &ATR );
        PLA_Obj_vert_join_2( A20, A21,  &ABL );
        PLA_Obj_view_all( A22,  &ABR );
     }
    < objects are freed>
} /*End of Program*/
```

Figure 4.9: PLAPACK/PFLAMBE code produced by the PLANALYZER

4.5 Chapter Summary

While computer code relies on what is underneath it, a “paper library” is not similarly dependent. Such a library assumes certain underlying functionality; it need not describe, down to the “bones” of the hardware, everything that must be done. This allows an expert in a higher-level domain to supply a library that needs to have its slots filled [57]. The traditional method supplies the pegs instead of the pegboard [5].

The important point is that a library either has to have the “right” level of modularity or multiple levels of modularity. Either avenue allows the user to program in a reasonable way, but it might be that only the latter situation really allows for machine-dependent optimizations to be carried out.

The automated code generation system described in this dissertation is an attempt to supply the “best of both worlds” to the user. The scripts would be considered underspecified and employing the PLANALYZER allows the automated coupling of this “paper library” to an underlying, encoded library.

Chapter 5

Automatic Analysis of an Implementation

This chapter presents an analysis strategy and a prototype implementation that utilizes the approach presented in this research work. This is important to the research presented here because the ability to determine the complexities and costs of algorithms is useful when constructing and maintaining linear algebra libraries.

First, the synergistic relationship between analysis and the design strategy, already presented, is introduced. Then the various "formats" of analysis are mentioned along with additional information regarding the parameters the analysis engine is intended to analyze. Finally, the potential interaction between the analysis tool and the algorithmic script language is discussed.

5.1 Motivation

Recall the example of Eager LU factorization illustrated in Figure 5.2. We consider the task of analysis by examining a script annotated with directives such as those given on lines 20 and 22-32 of that Figure. An example script may be seen in Figure 5.3, while an illustration depicting this chapter's place in the overall scheme of the document is depicted in Figure 5.1.

Notice that the script in Figure 5.2 makes only minor concessions to issues of implementation. The only indication that the script is intended for a parallel architecture lies in the annotations related to determining the size and data locality of `A11`. By way of contrast, the PLAWright code in Figure 5.3 not only contains directives that relate to the role of `A11` in matrix partitioning, but lower-level code that enforces where computation takes place by explicitly handling the communications involved. Further, that same script requires that a specific routine (`PLALu1`) be used to perform the local LU factorization and that the analysis engine should ignore what is in the performance section of the annotated library and apply the line-by-line performance measures included in the script.

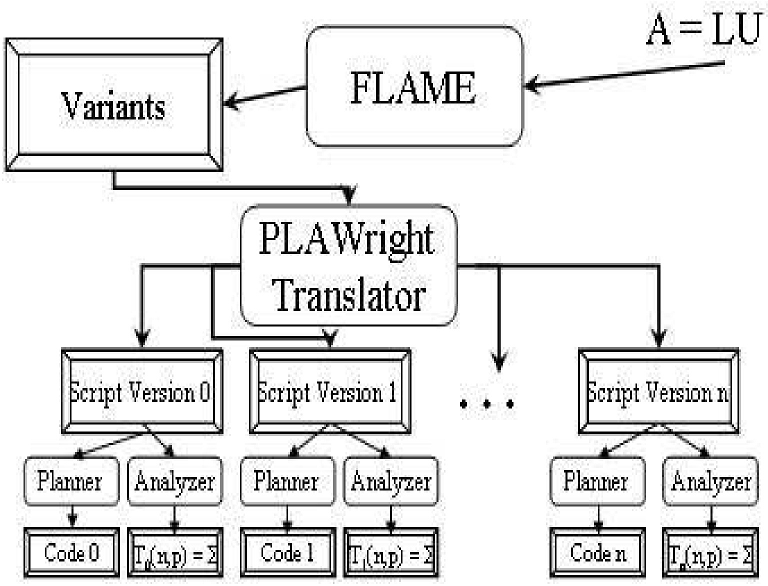


Figure 5.1: Where the analysis system fits into the "grand scheme" of things.

The task of analyzing the "simpler" script by hand involves a number of hurdles. First, one must determine what routines are involved. Then one must determine the performance characteristics of those routines. After one has determined such characteristics for each operation in the script, it is necessary to apply the analysis as the loop executes and the partitioning changes the size and shape of each object. While the application of line-by-line, annotated complexity estimation (as is seen in Figure 5.3) is also error-prone when done by hand, it does obviate the need to determine the performance characteristics of the routines involved. In either case, the task then becomes making the resultant formula useful in some manner.

There seems to be no escaping these problems unless one automates the process. Given an underlying library that is not "smart" (i.e. one that does not choose the best algorithm for the required operation), the simpler script forces the analyst to sort through all applicable routines in the library in order to determine the best routine available. An intelligent library attempts to pick the most efficient coding unit for each operation, but this makes the analysis task onerous because "the best" changes as the matrix sizes and shapes change throughout the course of execution. While the highly annotated script's analysis burden is unchanged, the *accuracy* of that analysis is questionable in this case because a great many simplifying assumptions are implicit in the per-line directives.

Therefore, automating the system of code production in such a way that the pro-

```
L has_property unit_lower_triangular ; // (* Permanent Property *)
U has_property      upper_triangular ;
A has_property square ; // (* Actually, Square here *)
L === A ; // {Recursive} {Permanent}  (* Establish name equivalence *)
U === A ; // {Recursive} {Permanent}
partition A =>    / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  where ATL is 0 by 0 ;
do until ABR is 0 by 0
   partition      / ATL # ATR \
                  |###########|
                  \ ABL # ABR /
                                  =>     / A00 # A01 | A02 \
                                         |#################|
                                         | A10 # A11 | A12 |
                                         |-----#-----------|
                                         \ A20 # A21 | A22 /
                  where A11 is local           and
                        A11 is locally square  and
                        A11 is nb by nb ; // No larger than is implied

   A11 = (L11\U11) <- lu_fact(A11) ;
   A12 = U12 <- L11^-1 * A12 ;
   A21 = L21 <- A21 * U11^-1 ;
   A22 <-  A22 -  L21 * U12  ;
   partition
                  / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  <=      / A00 | A01 #  A02 \
                                         |------------------|
                                         | A10 | A11 #  A12 |
                                         |##################|
                                         \ A20 | A21 #  A22 / ;
enddo;
L =!= A;
U =!= A;
```

Figure 5.2: Computer-readable script for Eager version of LU factorization

```
L === A ; // {Recursive} {Permanent}  (* Establish name equivalence *)
U === A ; // {Recursive} {Permanent}
L has_property unit_lower_triangular ; // (* Permanent Property *)
U has_property      upper_triangular ;
A has_property square ; // (* Actually, Square here *)
partition A =>    / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  where ATL is 0 by 0 ;
do until ABR is 0 by 0
   partition      / ATL # ATR \
                  |###########|
                  \ ABL # ABR /   =>     / A00 # A01 | A02 \
                                         |#################|
                                         | A10 # A11 | A12 |
                                         |-----#-----------|
                                         \ A20 # A21 | A22 /
                  where A11 is local           and
                        A11 is locally square  and
                        A11 is nb by nb ; // No larger than is really implied
   Performance performance_override("2*nb*nb*nb/3");
      function_override("PLALu1");
      A11 = (L11\U11) <- lu_fact(A11) ;
   EndPerformance;
   Performance performance_override("Bcast( nb * nb * 1/2) to PCC");
      Lower[L11tri] |* <-  Lower[L11] ;
   EndPerformance;
   A12 = U12 .<- Lower[L11tri]^-1 * A12 ;
   U11tri -* <-  Upper[U11] ;
   Performance performance_override("1/2 * nb * nb * Max(Length(Local(A21)))");
      A21 = L21 .<- A21 * Upper[U11tri]^-1 ;
   EndPerformance;
   Performance performance_override("Bcast(nb * Max(Length(Local(L21)))) to PCC");
      L21col |* <-  L21 ;
   EndPerformance;
   U12row -* <-  U12 ;
   A22 .<-  A22 -  L21col * U12row  ;
   EndPerformance;
   partition
                  / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  <=      / A00 | A01 #  A02 \
                                         |------------------|
                                         | A10 | A11 #  A12 |
                                         |##################|
                                         \ A20 | A21 #  A22 / ;
enddo;
```

Figure 5.3: Annotated script for Eager version of LU factorization

duced code and the produced analysis rooted in the same process is a promising avenue of research and it is detailed in this chapter.

5.2 Issues

5.2.1 Why Performance Is Important

It seems to be taken for granted that performance is important, but why is that? It is often the case that an individual does not need an answer immediately. Further, until Moore's Law runs out of steam, we are faced with an ever-faster array of processors. Thus, expending effort on optimizing code in order to improve performance by a few percent may involve unwise allocation of resources.

Certainly, this is a questionable practice if that optimization effort takes a great deal of time and has limited value. Chapter 3 sought to address the issue of programming ease and speed. If performing this optimization requires a small investment of expert resources, it may make sense to do so. In addition, it does not do much good to predict tomorrow's weather if the task is not completed until the day after tomorrow; some problems are such that they can take advantage of both the fastest machines and the fastest algorithms.

5.2.2 Why Performance Analysis Is Important

A basic question that may be asked is: "Is performance analysis necessary?" Obviously, it is not. There are many numerical libraries, both abstract and concrete, devoid of any analytical tools. However, there are drawbacks to that approach.

The first, and probably most important, disadvantage is seen when attempting to optimize such a library for a new architecture. Without formulaic guidelines it is difficult to predict how any given change will affect the performance of different parts of the executable. Similarly, it becomes difficult to determine where optimization efforts should be concentrated. One may be unable to readily determine if the problem lies in the algorithm or in a specific realization of that algorithm. Since there is no *systematically predicted* performance, there can be no "red flags" that indicate unexpectedly poor performance [37].

Predicting Performance

Trying to remedy systematic deficiencies by running a empirical tests is also an ill-considered approach. This method is time-consuming and tends to be resource-intensive. More important, the results of a large number of these tests may be required in order to determine what *parts* of the algorithms are responsible for cost overruns. While it may be possible to take a large amount of empirical data along with information about shared sub-components of the algorithms and use statistical analysis to determine where the bottlenecks are, it would be problematic to do so for at least two related reasons.

The first roadblock to this approach is the huge amount of data necessary for such an analysis when dealing with a large, monolithic library. There are simply too many variables to make this purely statistical method practical. The second problem is even more fundamental and difficult to overcome. Potential "feature interaction" would require that an exponential number of test cases be analyzed.

There are a number of sources for poor library performance, but all can be said to be in one of two major categories:

1. Routines with poor *predicted* performance.
2. Routines with performance that is poor (although *not necessarily predicted* to be so) [37].

It is not always the case that the hindrances can be classified as belonging exclusively to either category unless one employs a modeling strategy.

Determining the Sources of Performance Shortcomings

The algorithm itself is a potential source of inefficiency. As this is the core of an implementation, it can be the source of the greatest differences in achieved performance. Analysis tools may not construct a superior algorithm from an inferior one. However, they can be used to indicate the shortcomings in an algorithm and, possibly, to suggest algorithmic changes that will result in superior performance. These clues may result from contrasts between two algorithms intended to perform the same task, or from a mismatch between the performance that the user predicts, based on experience with similar algorithms, and the performance predicted by the analytical engine (with its built-in knowledge of the underlying algorithmic and architectural interactions).

It is not surprising that the implementation of the algorithm can be the source of variations in performance. There are some potential sources of inefficiency that apply only to the parallel computational case, while others apply to both the serial and parallel instances. These sources include the use of improper communication algorithms, a mismatch between theoretical models and real machines, and unfortunate assumptions about the use of processor and memory resources and their interactions.

We note that it is sometimes difficult to determine when the performance failings are the result of poor algorithmic design or implementation details. For example, if one takes a high-level view, it is possible to predict superior performance in an algorithm. Yet, one may know that the algorithm will translate into an implementation that has poor performance regardless of the real machine used. Alternatively, this poor performance may be completely dependent on the details of the underlying computational system.

Code Steering

We wish to have the PLANALYZER select the "best" algorithm in a given situation, but we also wish to equip the end-user/programmer with the ability to guide the system to a routine/method that he believes is better (or wishes to study). Therefore, whatever methods are used, (incremental) user-interaction should be kept in mind even if the software does not present a "point-and-click" type of interface.

5.2.3 Convenience vs. Performance

The analysis framework and tools should help to assuage a typical fear about scripted languages. Namely, that they are convenient to use, but their performance tends to be poor. Accepted wisdom holds that trying to retain this ease of expression as one migrates to a parallel environment is likely to exacerbate these problems. One can find any number of examples where this "rule of thumb" does, in fact, hold true [20].

The development system presented here attempts to address both convenience and performance concerns. Allowing this freedom is an effort to strike a balance between too much and too little guidance being provided by the software. It is made possible by making the ability to cleanly mix the layers of annotation and scripting a central concern.

There are a number of ways in which this work deals with performance considerations. We assume that the underlying library (the target of script translation) is made up of efficient routines. Therefore, a script translated into a set of calls to that library should also be efficient.

If the performance of the existing code segments is analyzed properly and if a systematic way of gluing them together intelligently to perform the new algorithm can be constructed, high-performance should be achieved. Here, "high" is defined to be as performance comparable to that which someone intimately familiar with the underlying library could effect.

User Benefits

The potential benefits yielded by our analysis tools, largely mirror those of handcrafted analysis. Analysis tends to provide guidance for algorithm and implementation tuning along with information regarding case-specific proximity to optimal performance.

While this sort of activity can be done by hand, it is made much easier by computer assistance in a number of ways. First, when one is dealing with a large library, the individual analysis tasks are time-consuming. The determination of relationships and interactions between routines is more so. In addition, from a psychological point of view, this activity requires shifting back and forth between different concerns and that tends to impose an even greater time penalty on the designer.

The most obvious benefit to analysis tools is the ability to quickly and dynamically determine the complexity of a given algorithm or implementation. This allows the designer to determine the efficiency of the algorithm at various levels of detail. One does not have to waste time tuning an algorithm of inherently sub-optimal complexity. Further, when dealing with a multi-tiered algorithm, the analysis may reveal patterns across and interactions between different levels and modules.

While the analysis system may not suggest solutions for unnecessary interactions, couplings, and dependencies, it can make them obvious to the experienced designer and more apparent to the novice. In a similar manner, the analysis system may reveal cases where the specificity of the situation is not being taken advantage of by the designer.

User Responsibilities

The responsibilities of the motivated user who wishes to exploit all of the abilities of the analyzer are too situation-dependent to be detailed here. This section, instead, gives and introduction to the features and requirements of the system as they relate to the "casual" user.

The user must supply input to the analyzer in a form that the analyzer can read. However, the programmer need not be concerned with how heavily annotated his scripts are because the output *form* of analysis is not entirely dependent upon the form of the input.

The other matter is the specification of the output. There are many potential forms that output might take. While there are default settings, the PLANALYZER also allows for the specification of different ways in which to measure (e.g. operation counts, time taken etc.), different forms of expression, and exactly what to measure (e.g. communication time only).

5.2.4 Traditional Approaches

Typically, algorithmic analysis in this area has been both manual and somewhat *ad hoc*. The usual scenario involves the analysis of an algorithm as a stand-alone example. The reasons behind performance differences in variations on an algorithm are largely hidden because of the monolithic nature of the analysis.

5.2.5 Problems with Traditional Approaches

While such an analysis may be accurate, it is not as useful as it might be. Without a systematic approach to the analysis of a family of algorithms, it is difficult to determine the comparative advantages and disadvantages of the algorithms. Specifically, this approach is of severely limited value in the construction of hybrid or polyalgorithmic variants [40, 56].

5.2.6 A New Approach

Given a systematic approach carried through the design of a library, one can analyze algorithms that rely on the components of that library. It is the interaction between levels of the library that tends to make this analysis difficult. A consistent approach in library design leads to a consistent pattern of interaction.

Research into the issue of hybridization [40] gave us some insight into how useful the systematic construction of the algorithms and the layering of the library were when it came to accurately modeling the target computational environment. Preliminary tests showed that these analytical models were reasonably accurate.

This systematic nature also provides for the construction of *automated* analysis tools. These tools allow for a more systematic and informed approach to the optimization task that is typically so onerous in the absence of a unified approach, let alone such an automated tool. The central idea is that the performance annotations mirror the code that, in turn, mirrors the algorithm. Thus, to use the tool, one only need be an "expert" in the construction of algorithms.

By compiling the algorithmic script into both a functional program and an analytical code readable by the *Mathematica* [77], symbolic manipulation package, one can interactively develop and analyze these algorithms immediately, in the same, automated environment. Further, these analyses need not be tied to a single set of expressive primitives, such as time required, but may be re-formulated in terms of operation-class counts, etc.

5.2.7 Coupling Code and Performance

The module-dependency graph of a systematically constructed, layered library has fewer leaves than that of a haphazardly constructed library providing comparable functionality. If we implement our own communications library in terms of some set of primitives, we have more control and fewer microbench tests to perform. The same approach can be extended to a very low level, but there is a trade-off. We must determine how sophisticated to make the code $\rightarrow$ performance parser and the right balance to strike between readability, accuracy, and work-intensity. Annotating the library at too high a level, results in accuracy at the cost of having to benchmark and annotate too many routines. Doing so at too low of a level makes the intermediate form of performance code difficult to simplify. It is logical to make the annotations look like code to as great an extent as possible so that both are readable and so that it is not necessary to learn a new "language" for each task.

Library Strata

One of the most basic reasons for the requisite flexibility of the modeling strategy is that what comprises an "operation" changes as one proceeds in designing, implementing, and refining an algorithm. For the tool to be useful it must be able to address the needs of the designer as his view of the operations changes. While this can be motivated in the sequential arena, it is more straightforward to do so in the context of a parallel environment.

Consider a simple algorithm like the outer-product computation that was discussed in the LU decomposition algorithm. Obviously, in the distributed case there are a number of ways to define what it means to perform a matrix-matrix multiplication. For instance, there is the entire multiplication: $A_{22} \leftarrow A_{22} - \vec{a}_{21}\vec{a}_{12}^T$. Even if we ignore details of implementation, we may consider the time spent performing the calculation to be restricted to the time spent doing so on a given processor. We may wish to ignore time taken to perform the manipulations involved. Further, we may consider some of the implementational issues that arise as part of the SUMMA algorithm. We may wish to perform the matrix multiplication with a set of columns (e.g. A_{21} instead of $\vec{a}_{21}$) in which case "the multiplication" may be any of the component multiplies, global or local, of this larger multiplication. Therefore, the analysis system must allow a shift between these different views.

Independent of the form the analysis takes, two fundamental questions must be answered:

1. What qualities are to be analyzed?

2. In what quantitative terms should these qualities be expressed (i.e. what are the "units" of analysis)?

In the area under study, the answers to these questions are readily available. The analysis system measures the time and memory required to perform a given algorithm. Such qualities have generally accepted unit-measures; time is generally measured in CPU (milli-/micro-) seconds while memory used is measured in (kilo-/mega-) bytes.

While these two answers provide all that one may *require* from a system geared to purely *practical* analysis, the features that they enable may not be sufficient for a flexible analytical tool for a number of reasons. The most basic difficulty is that these measurement quanta may not allow the measurements to be expressed in a manner desired by the user. For example, if one wishes to determine the number of matrix-matrix products that are performed, time and space complexity measures may not necessarily yield useful information. However, intelligent structuring and base-level specifications yields a set of constructs that can be used to express both. Further, there are guidelines that help one to determine the kinds of primitives that must be provided if a certain kind of feedback is desired.

Parameters of Analysis

One should be able to use case-specific information during the analysis of an algorithm. Certain measures have no meaning if one does not have a machine *model*, but do not require a machine *instance* in order to be defined. Other measures require a fully-specified machine (and problem) environment in order to have meaning. Given these facts, the analyzer is designed around a set of primitives that yield great flexibility in these areas. Furthermore, to facilitate feedback in the desired format, the underlying language should provide for the dynamic (user-based) creation of new "concepts."

Let us be more concrete. The useful object-based abstractions under consideration: manipulation, calculation, and property determination, have already been discussed in Chapter 3. Almost any non-trivial algorithm uses all of these abstractions. Therefore, the analysis must involve, or *allow the involvement of*, all three. The caveat in regards to *allowing* the inclusion of measures for some abstractions is included as one may also wish to ignore certain measures. Most obviously one might wish to discount property determination as this calculation is often computationally trivial. Further, one might wish to ignore manipulation time and space complexity. Alternatively, when one wishes only to consider scalability issues, it is often convenient to ignore everything *except* the time spent in the manipulation (communication) subsystem. It is not difficult to create other cases wherein one might wish to consider only *parts* of some of the abstractions while ignoring others.

There are many ways to construct the framework of this analysis system and the implemented computational engine. It seems necessary to allow the user a great deal of control over the primitives and concepts composed from those primitives. However, it would seem that there should be a certain "default" setting that is both *flexible* enough to provide a tool for users with many disparate needs and *conventional* enough to provide feedback in a format that is commonly seen in papers on the analysis of similar algorithms. The primitives provided should be useful in a wide range of analysis tasks. This is because the extension of the PLANALYZER through the inclusion and definition of new primitives requires greater expertise than is practical to expect.

While it may seem a bit confusing to mix terminology with regard to analyzing *algorithms* and analyzing *programs*, perhaps it should not. In a distributed computational environment, it may be possible to ignore the model versus implementation distinction. It is probably most useful to think of physical computational systems as somewhat complicated models. This is not a new idea; any system can be mimicked with a complex enough model via successive refinement. This dissertation focuses on providing a useful model as well as a systematic way to determine a base set of primitives that have to be evaluated so as allow the determination of fully quantified results. We are concerned with the clarity with which the tool under consideration here supplies information. However, the goal of automating the kind of performance profiling that has traditionally been done by hand is also a consideration.

Analysis of Components

In order to perform analysis by composing "building block" analytical modules, some base level of analysis must be determined. The simplest form of composition would be the unadorned addition of these components (formulae). In this section, we assume that this is how analysis is carried out. Later sections discuss why this simple approach may be insufficient.

The previous section discussed some of the issues that need to be considered in the construction of the analysis tool. Among these was the determination of what is to be measured, in what terms that measurement is to be expressed, and what makes up the primitive set. Let us, for the moment, restrict ourselves to a small but useful set of measurements; the α, β, and γ time-complexity set. Here, α is the start-up cost for a message, β the cost per item sent, and γ the time per computation. This is a simple view, specialized for the distributed computing case. However, there are analogies to α and β in a serial architecture, and multiple γs can be used, so this model is useful.

The next task is to determine which components must be measured. The last section discussed why this is a question. Let us suppose that we have made a utilitarian decision. If we wish to analyze a library, we can express the lowest layer (the leaves) in terms of the primitive measures (the α, β, and γ mentioned previously) and describe the other layers in terms of those beneath them. There is a trade-off between accuracy and annotative expediency with this approach favoring the latter.

While the assumption is that the library is layered, this is not strictly necessary. Many modern software packages, such as Sniff+ [12], automatically determine the calling structure of a set of routines. From this directed graph, it is possible to construct a complexity model from the leaves "in." While this situation is not optimal, it does not present an insurmountable block to the analysis strategy discussed in this chapter.

One problem that may occur to the reader involves the modeling of the leaves. The leaves do not rely upon any other (visible) routines. Typically, one performs empirical measures on these components for various problem and computational grid sizes and then uses something akin to a line of best-fit to express their complexity. These routines often have performance characteristics which are dependent on problem-specific details such as operand shape. The user needs to determine the level of accuracy that they require of the analysis system in order to determine how highly refined the base-level analytic models need to be.

Synthesis of Component-Analysis

We assumed that the analysis of a component that utilizes other analyzed components as building blocks was a solved problem. Let us consider the fact that we may eventually wish to simplify the resulting analyses. In that case, to analyze a component it may be beneficial to synthesize the analysis of the sub-components which make up the routine (component) to be analyzed.

The most obvious application of "synthesis" is the simplification of the implicit summations that occur over a looping construct within a routine. Once the summation is made explicit, simple mathematical substitutions can be made to reduce the complexity (as measured by lexical length) of the expression.

It should be pointed out that this synthesis is not always a good idea. For, if one performs the synthesis at the lowest level, it may be considerably more difficult to combine expressions at higher levels without sacrificing accuracy.

5.3 Contributions of the Systematic Underpinnings

Approaching the design of linear algebra algorithms in a systematic fashion reduces the difficulty of the analysis task. Our approach to algorithmic and library construction tends to simplify and make explicit the relationships between different parts of the programs as they relate to overall performance. Often, implementors optimize algorithms in a compartmentalized fashion. They rely on intuition and experience rather than complexity measures to drive their optimizations and tend to view each improvement without considering its impact on the larger picture.

Perhaps this is almost unavoidable when the routines to be optimized are parts of a library with no underlying framework. The analysis required in such a case could be monumental. There are two major roadblocks to be considered:

- Monolithic construction methodology and
- Modular, but poorly thought out, construction practices

If the library is modularized, the different routines tend to call on one another. However, modularity does not imply design soundness, and these relationships between modules may not follow any discernable pattern. The combination of these two properties complicates the analysis task. The monolithic alternative may seem preferable as that strategy avoids the complications caused by module interactions. Unfortunately, that approach yields a new analysis task for every derived algorithm and fails to provide any sort of framework from which to gain leverage from the analyses already performed. Not only does this result in more work for the analyst [70], but it also seems to disallow even the *possibility* of determining meaningful patterns unless the specification of the sub-components is systematic.

Conversely, if the software system is built with a unified approach and utilizes a systematic methodology to build the algorithms, not only is the construction process eased, but the analysis is considerably less complicated. The design process allows one to follow the framework of the supplied algorithms. Since analysis tasks can mirror the structure of the objects of their analysis, they can be constructed top-down, bottom-up, or middle-out along with those algorithms. It should also be noted that the algorithmic design could follow the analytical work.

Many of these benefits come "for free" when the modularity of the software is presumed to be logical and easily understood. However, most of them are simply *enabled* by this systematic construction. There is still something of an onus on the (low-level) designer to specify the functionality, complexity types, parameters, and measures to the analysis engine, but it should be noted that:

1. The layered construction, in concert with the FLAME methodology, eases the determination of the patterns seen in a given algorithm and

2. The formulaic specification of these patterns opens the door for a systematic classification of these patterns [43]).

5.3.1 Modularity of the Analytic Harness

There has already been considerable discussion about the various uses of and advantages to an integrated analysis strategy and system. This section attempts to point out the differing impact that such tools have on various types of libraries.

One must consider the manner in which a designer would interact with the design system. Thus, the first subsection deals with issues related to hand-built software systems as well as presenting some synthesis of the relevant ideas already discussed. The next subsection deals with the more pertinent ideas in relation to automated library construction. Given the cookbook nature of the algorithmic construction and analysis, systemic automatization appears to be a realizable goal.

Impact on Manually Assembled Systems

Typically, a library, even if constructed in a very systematic way, is hand-written by programmers (or non-programmers in the case of "paper" libraries mentioned in Section 4.5). Since this approach to library construction is the one most applicable to both well designed and poorly designed libraries, let us consider what can be done in the latter case (as the former has much in common with the automated situation discussed in Section 5.3.1).

While the well-integrated, flexible analysis tools discussed here are not entirely amenable to use in a "disorganized" environment, it might be possible to gain some advantage from them. If the analyst is willing to delve into the particulars of the composition and analysis structure, it may be possible to regain *some* of the flexibility possessed by the tools in the more well organized case.

The first assumption is that the code to be analyzed is neither written in the script language nor in a style that mirrors that language. This assumption is made because if it is written in that style, the analyzer can be used on the script or on something that can be reverse-engineered from the code.

The easiest way to use the analysis tools in this case would be to hand-translate the given code into the corresponding script. One might have to translate a number of routines into the script language before getting meaningful feedback from the automated system. However, the user might wish to declare the routines themselves as primitives, or use the analysis engine's abilities to redefine "concepts," and supply their own complexity measures for the routine.

This approach is may result in analyses that lack comprehensibility or fail to reflect algorithmic modifications. Both of these problems can be ameliorated to some degree if the user is careful in their design of primitives and concepts, making them compatible with the remainder of the automated analysis engine. While it may be that the engine lacks some of its former ability to simplify the resultant equations, little should be lost in terms of reflecting algorithmic changes if the user is careful to provide layers *similar* to those discussed here. The analysis engine should also be modular and layered as is the case with the prototype under consideration in this chapter.

Impact on Automated Systems

We now begin a discussion regarding how the analysis engine may aid the automation process and how automation makes the analysis chore simpler. At the same time, we need to address what is required of the user.

Given an automatic tool for the construction of these algorithms, this system might be used to hybridize algorithms already instantiated. Given an algorithm for computing function A using method I, the system presented in this dissertation could generate methods II and III. Each method has its advantages and disadvantages. Often determining when one algorithm is superior to another is a complex task. Given an engine that generates equations that can be evaluated on the fly, such hybridization would become mechanized. This same approach could prove useful in the case that several levels are simultaneously hybridized. However, it becomes less reasonable to ignore evaluation (selection) time as one goes down to lower levels of the memory hierarchy.

Many of the issues relevant to the analytical tool are independent of this generator. Such a tool could be used to select the "best" algorithm from a library, even when that library has nothing to do with the system described here, provided that some sort of "handshaking" between requirements and provided services [30] can be performed. If the system can determine that a given routine fulfills the requirements of a given "call," then the system could take pre-evaluated information about these "gray box" routines and determine which variant is optimal in a given situation.

5.4 Implementation: Automated Analysis

Thus far we have discussed what is desirable in the abstract. Now, we delve into issues of implementation. To review the current stage of the process as it now stands, the reader is referred to Figure 5.4. In particular, the reader's attention is directed to the two boxes in the lower-right quadrant of that Figure.

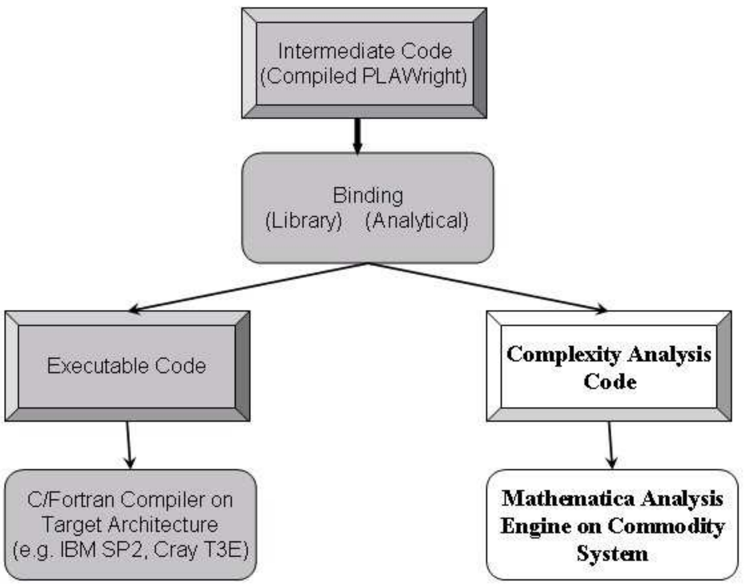

Intermediate Code (Compiled PLAWright)
Binding (Library) (Analytical)
Executable Code
Complexity Analysis Code
C/Fortran Compiler on Target Architecture (e.g. IBM SP2, Cray T3E)
Mathematica Analysis Engine on Commodity System

Figure 5.4: The position of the analysis engine in the context of the implemented system.

5.4.1 An Analysis-Ready Script

Let us consider a script presented in the preceding chapter, Figure 4.4 (page 75). Recall that this algorithm is a version of the Eager variant to LU factorization. In that script, the user explicitly controls the data distribution so that only local computations (computational kernels) are required. In Figure 5.5, a replication of Figure 5.3, two lines (25 and 28) of this script are annotated with their associated cost.

5.4.2 Explanation of Script Extensions and Line-Cost Estimates

A few questions may arise upon viewing this annotated script (Figure 5.3). For example, one might ask why some of the lines have no associated cost. This script reflects a somewhat arbitrary decision. The rationale is that those operations that have a cost which does not

```
L === A ; // {Recursive} {Permanent}  (* Establish name equivalence *)
U === A ; // {Recursive} {Permanent}
L has_property unit_lower_triangular ; // (* Permanent Property *)
U has_property      upper_triangular ;
A has_property square ; // (* Actually, Square here *)
partition A =>    / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  where ATL is 0 by 0 ;
do until ABR is 0 by 0
   partition      / ATL # ATR \
                  |###########|
                  \ ABL # ABR /   =>     / A00 # A01 | A02 \
                                         |#################|
                                         | A10 # A11 | A12 |
                                         |-----#-----------|
                                         \ A20 # A21 | A22 /
                  where A11 is local           and
                        A11 is locally square  and
                        A11 is nb by nb ; // No larger than is really implied
   Performance performance_override("2*nb*nb*nb/3");
      function_override("PLALu1");
      A11 = (L11\U11) <- lu_fact(A11) ;
   EndPerformance;
   Performance performance_override("Bcast( nb * nb * 1/2) to PCC");
      Lower[L11tri] |* <-  Lower[L11] ;
   EndPerformance;
   A12 = U12 .<- Lower[L11tri]^-1 * A12 ;
   U11tri -* <-  Upper[U11] ;
   Performance performance_override("1/2 * nb * nb * Max(Length(Local(A21)))");
      A21 = L21 .<- A21 * Upper[U11tri]^-1 ;
   EndPerformance;
   Performance performance_override("Bcast(nb * Max(Length(Local(L21)))) to PCC");
      L21col |* <-  L21 ;
   EndPerformance;
   U12row -* <-  U12 ;
   A22 .<-  A22 -  L21col * U12row  ;
   EndPerformance;
   partition
                  / ATL # ATR \
                  |###########|
                  \ ABL # ABR /  <=      / A00 | A01 #  A02 \
                                         |------------------|
                                         | A10 | A11 #  A12 |
                                         |##################|
                                         \ A20 | A21 #  A22 / ;
enddo;
```

Figure 5.5: Optimized script for Eager method of LU factorization with performance annotations

depend on the size of the object and are low enough so as to be considered "noise" are ignored and others are assigned the complexity measures corresponding to the performance annotations provided by the library. Our focus is on the on the critical path of execution and those functions which contribute to it. Thus, global operations are the items of greatest import and receive the most attention in the analysis phase. The second easily motivated question regards the line-by-line cost assignment. One could have assigned a cost to the entire script or to every do-enddo loop as both are viable alternatives. However, the analysis issues that arise are more easily motivated by this line-by-line cost-assignment technique.

Given the annotated script and the summation expression reflecting the cost of the script (seen in Section 5.4.3), a few questions arise. The two that relate to the annotations themselves are the most easily dispensed with. The Max(Width/Length(Local(object))) is simply a functional programming notation for determining the maximum size of the object in a given dimension over the set of nodes (i.e. how much is held by the node that holds the most). This is done because this maximum tends to be the bottleneck for the algorithm. The second is the "Broadcast" function. This can be replaced "underneath" by any method of broadcast and the analytical annotation reflects the complexity of the algorithm employed.

The expression reflecting the cost of the algorithm embodies a number of implicit assumptions. While these assumptions are not strictly enforced in the analysis engine, they are useful in order to present a simple example. As was mentioned above, the Broadcast may take place in a number of ways. Therefore, its cost depends on the machine architecture and the manner in which the broadcast is performed. Here, for simplicity, a two-dimensional mesh is assumed, and the broadcast proceeds via a minimum spanning tree algorithm. While this convention regarding the broadcast is logical and not greatly limiting, the second simplifying assumption is a bit more restrictive. In order to present a concise summary formula, we have assumed three things:

1. That the distribution blocking size is the same as the algorithmic blocking size.
2. That the size of the matrix (n) is an integral multiple of this blocking size (nb).
3. That we have used a block-cyclic distribution in both dimensions.

In Section 5.6 these restrictions are relaxed. In such cases, accuracy tends to come at the cost of intelligible cost expressions.

5.4.3 Analytical Result

Computing the total time required for the parallel LU factorization, $T_{\mathrm{LU}}(n, r, c, b)$ thus requires us to evaluate

$$\begin{aligned} T_{\mathrm{LU}}(n, r, c, b) = \sum_{i=1}^{n/b} \Bigg[\frac{2}{3} b^3 \gamma \quad &+ \quad T_{\mathrm{bcast}}(b^2, c) + b^3 \lceil \frac{n-ib}{cb} \rceil \gamma + T_{\mathrm{bcast}}(b^2, r) + b^3 \lceil \frac{n-ib}{rb} \rceil \gamma \\ &+ \quad T_{\mathrm{bcast}}(b^2 \lceil \frac{n-ib}{cb} \rceil, r) + T_{\mathrm{bcast}}(b^2 \lceil \frac{n-ib}{rb} \rceil, c) \end{aligned}$$

$$+ \quad 2b\lceil\frac{n-ib}{cb}\rceil\lceil\frac{n-ib}{rb}\rceil\gamma\Big]$$

where b equals block size `nb`, r and c are the row and column dimensions of a two-dimensional processor grid, i equals the iteration index, $T_{\mathrm{bcast}}(m,p)$ equals the cost of broadcasting m items within p processors, and γ is the cost of a floating-point operation.

While this expression can be easily evaluated, given a cost estimate for the broadcast, it is typically useful to have a more compact estimate for the cost. For example, if one wanted to dynamically choose between different implementations, a cheap estimate of the cost must be available. Derivation of such an estimate is straightforward, but tedious and error-prone if done by hand. Thus, we have created a prototype system employing *Mathematica* that can take the script input and generate a cost estimate that is compact in form. However, this estimate may not be of great informative value.

5.4.4 The Use of*Mathematica* `Module[]`s

Thus far, the performance characteristics have been discussed with little specificity about what the annotations include or what form they take.

Since the focus of the discussion is limited to imperative languages, such as Fortran and C, it seems that the level of the subroutine or procedure call is certainly the most convenient location in which to place this annotative information. It should be pointed out that functional supply (what the routine furnishes) and performance characteristics are two separate ideas, but can both be viewed as meeting the requirements of a programmer. Further, it is important to note that various language constructs (selectors, loops, etc.) can be thought of as meta-subroutines. Combining a loop with a routine creates a new routine with different performance characteristics; characteristics that are calculable from the two components involved.

5.4.5 Performance Estimates: Discrete Formulae

Discrete formulae arise from the analysis of the algorithms under study in this document. As can be seen in Section 5.4.3, one possible analysis format is the result of summing together all of the individual operation counts on a per-loop basis.

Why Discrete Formulae Arise

All commonly used modern computer architectures are discrete. It should not be surprising that a model of these systems gives rise to discrete mathematical formulae.

Algorithms from the area of linear algebra, can also give rise to discretized equations when one describes their complexity.

Problems with this Format

As we see above, the expression that results is somewhat unintelligible, cluttered as it is with summations and ceiling functions. Such results tend to be difficult to interpret. They are also poor formats for determining performance *profiles*, especially when many parameters may be varied simultaneously.

5.4.6 Closed-Form Expressions

While discrete analysis allows for accurate modeling, it tends to fall short in presenting the user with understandable information. Typically, the lexically shorter approximations are worked out by hand. The constants involved are tedious to calculate for various machine architectures. In order to do so, it is often the case that a number of simplifying assumptions are incorporated. It is sometimes the case that these assumptions have a great impact on the reliability of the resulting formula. It is our goal to design the analysis system so that the task is eased and this impact is minimized.

A Numerical (a.k.a. A Statistical) Treatment

Of course, a number of data points from discrete analysis can be taken as guides for such things as a least squares fit to a function of a known degree and form. While determining this degree is not always simple, it is usually reasonably straightforward because of known algorithmic complexity properties. Using modern tools such as *Mathematica* or Matlab, the difficulty is less in the determination of a line of best fit than in giving meaning to the coefficients that describe that line. The current state of the PLANALYZER system is such that that these equations can be generated, but the coefficients have no explicit connection to the parameters of the procedure analyzed.

Highly Simplified Models

Because the analytical system is symbolic, it is relatively straightforward to generate closed-form results by sacrificing accuracy. For example, instead of computing the time taken to perform operation X, the analytical engine can count the number of times operation X would be called and produce a result of the form #X. The same idea can be used to yield counts of different categories of functions, counts of functions that run at some percent of the processors peak rate, etc. While this form is not what is typically referred to as "closed," there are cases where this might provide more useful information to the developer. For example, if the programmer is attempting to move operations from level-2 to level-3 BLAS, it would likely be beneficial to determine if various changes to the code were having the desired effect. The method outlined above would automate that process.

5.4.7 More Practical Concerns

Some issues only have a place when the discussion is grounded in implementation. Those issues are presented, briefly, here.

Viewing the Processor Set

There are two ways that one can view the processor set when it comes to the analysis of an algorithmic implementation:

- unified and
- component-wise

The view of the processor set as unified ignores the individual differences between the processor's work sets as well as any difference between the component processors. The latter simplification may be considered harmless because heterogeneous computer systems are not considered in this document because of the complexity that their design inflicts on any such analysis [13].

There are a number of approaches to unified processor modeling. The approach used in the prototype system presented here could be called "single-case" based. The PL-ANALYZER determines the best/worst/average complexity during any given step of the computation (where a step may be defined to any level of granularity) and sums up these steps, in whatever manner, to yield the result. Many other approaches are possible. One such approach would be interval-based. Such a system keeps track of a set of cases (e.g. best and worst) and calculates not a single cost, but the interval over which the costs may range. The approach that we selected seemed capable of providing the information required and is more typical of the analyses traditionally seen in the area.

The single-case based model also appeared to be the most appropriate as our interest was in constructing a proof-of-concept system that addressed the complexity of the critical path of the code/architecture under consideration. Therefore, modeling those algorithmic steps that would likely prove bottlenecks in the execution of the code was the foremost concern. As can be seen by studying the results presented in Section 5.6 this strategy can yield highly accurate results when many operations are global and involve collective communications. In such cases, determining the steps along the critical path can be done via the use of a model that lacks much of the detail that would be required to mirror the underlying library with total accuracy.

5.4.8 Load Balance

The analysis scheme should have the *ability* to deal with load balance. This is not to say that it should do anything about fixing existing load imbalances past revealing them to the designer.

The term "load imbalance" is typically taken to mean raw computational imbalance. In other words, different processors have different operation counts. This is a valid interpretation of the term, but the meaning of the term can be extended in a number of ways.

One of the chief sources of optimization difficulties is the insufficient refinement of processor timing differences. While very high-level abstract machine models do not evince operation speed differences, useful ones usually do. Therefore, the analyzer must model not only the number of (basic) operations done, but also the (relative) speed at which the target architecture is capable of doing them. This can be done by a very detailed modeling of the underlying architecture, specifically the memory hierarchy and timing, or through the creation of a base set of operators that facilitate the exposure of these timing imbalances. The work outlined here takes the latter approach; championing the use of a (flexible) framework so that these different "kernel" rates and complexities may be specified.

In addition to allowing the proper level of performance resolution, the analysis system requires the ability to refine the view of the processors. It is important to note that this does not mean that the analyzer must "imitate" the processors in a lockstep fashion. As in the case of the kernel complexities, it is important that the design system allow the user to tune the specificity of their input to match the detail level that they require in the analysis system's output for at least two reasons. First, it requires extra work to provide succinct information when the analysis engine is provided with a highly detailed system "map." Second, it is impossible for the analysis to provide highly accurate feedback if the information provided is at too high a level. The latter is not surprising, but it is important that the former be pointed out because it often takes computational and programing effort for an automated analysis tool to disregard information provided to it.

5.5 Related Work

Many of the papers in this area are almost exclusively empirical in their treatment of the presented algorithm(s) [33]. Such work presents an algorithm then discusses various issues that revolve around a coded instance of the algorithm under consideration along with some real-world experimental (timing) results. Often, work that is more scholarly discusses the presented algorithms in terms of such things as complexity measures. These are often followed by empirical results as "proof" of the correctness of the more abstract resultant formulae [24, 70, 31, 53].

5.5.1 Monolithic Analysis

The analysis of individual routines is often done in something of a vacuum. Usually, this approach is taken when one's goal in analyzing an algorithm is to obtain maximum accuracy. By viewing the algorithm under consideration as a unit, all of the computational issues can be tackled in order to yield an accurate reflection of the performance of the algorithm. The downside of this approach is that it gives little leverage for tackling the next analysis task.

5.5.2 Ad-hoc/Component Sums Based Analysis

At the opposite end of the spectrum is the component-sums approach to analysis. This approach simply glues together the results of the analysis of the pieces comprising the overall algorithm. This allows for the rapid synthesis of analytical components, but the manner in which these components interact is not modeled and often difficult to determine.

5.6 Experimental Results

In Section 4.4, a number of variants and versions of the LU factorization algorithm were presented along with a discussion regarding the code generated by the PLANALYZER. In this section, the same intermediate-language form that is translated into C code is instead transformed into a form of code that serves to model the performance characteristics of the resultant program as it executes on the target architecture.

5.6.1 Automated Analysis Generation

The analysis presented in this experimental section is numeric, not symbolic, in nature, as it would require a good deal of analysis effort on the part of the author to determine whether the analysis was correct in the latter case. In order to evaluate the accuracy of the performance estimates generated by the analysis engine, it was most expedient to compare the numerical estimates generated with the witnessed empirical performance on the target architecture.

In essence, the analytical engine works by executing the analysis scripts that are generated along with the executable code. The performance estimates for the leaves of the PFLAMBE software architecture were the result of a great deal of experience with the code-generation system and the computational environment under study, but were not as precise as benchmarks would have been. However, this level of detail would allow for a more rapid alteration of the analysis engine so as to produce symbolic results, so was left as is. As we can see in the next subsection, the estimates accurately reflect the performance of smaller problems as well as illustrating performance trends for each of the cases examined.

5.6.2 Analysis vs. Witnessed Performance

In all cases of comparison between estimated and witnessed performance included here, tests were performed on an 80 node Cray T3E (lonestar.hpc.utexas.edu). While the algorithms would have run on non-square computational grids, only square grids of sizes 2×2, 4×4, and 8×8 were tested. The same tests were performed in all cases with a few provisos. The global size of the (square) matrices tested ranged from order 32 to order: 4096, 8192, and 16384 for the 4, 16, and 64 node cases, respectively. However, due to resource limitations, some of the computationally inefficient algorithms were not tested with the largest matrix sizes. The analytical system would have predicted the timeouts that occurred (one is given a maximal allotted time when one submits a job to the T3E), but it was not used for this purpose.

First, let us examine the predicted and witnessed performance of the five variants listed in Section 4.4. These results are depicted in line-graph form in Figure 5.6, Figure 5.8, and Figure 5.10 and in bar chart form in Figure 5.7, Figure 5.9, and Figure 5.11. The shade of the bar indicates the quality of the estimation, with black being used if the estimates are more than 20% off, gray for 10%-20% off and white for an error of less than 10%.

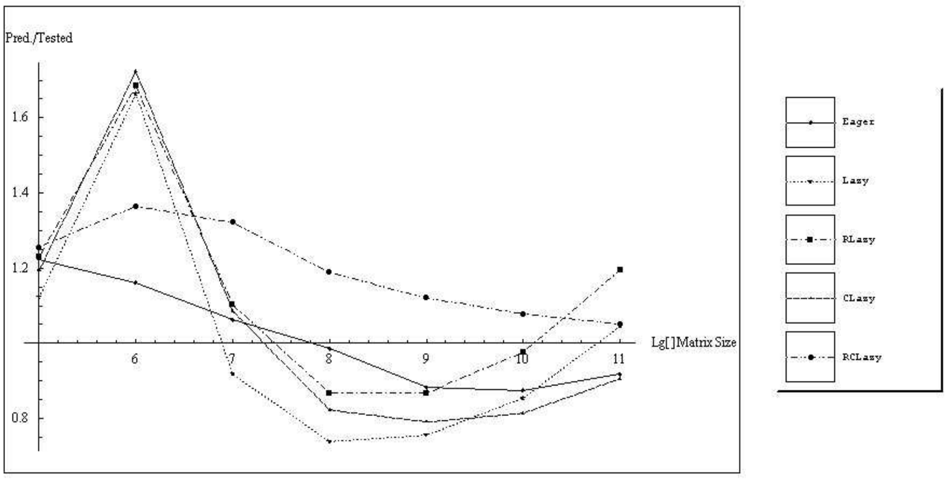


Figure 5.6: Ratio of predicted to achieved performance: 4 node Cray T3E

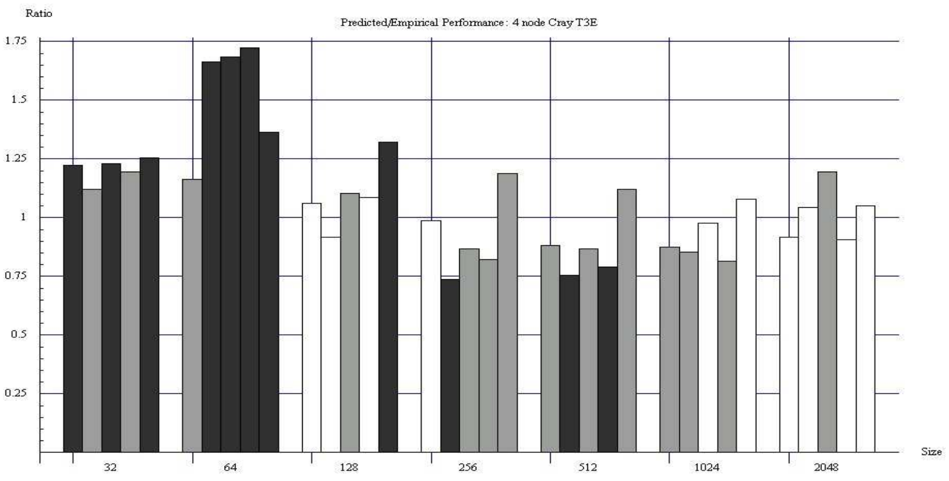


Figure 5.7: Bar graph indicating ratio of predicted to achieved performance for 4 node Cray T3E. From left-to-right the bars correspond to the Eager, Lazy, Row Lazy, Column Lazy, and Row-Column Lazy implementations.

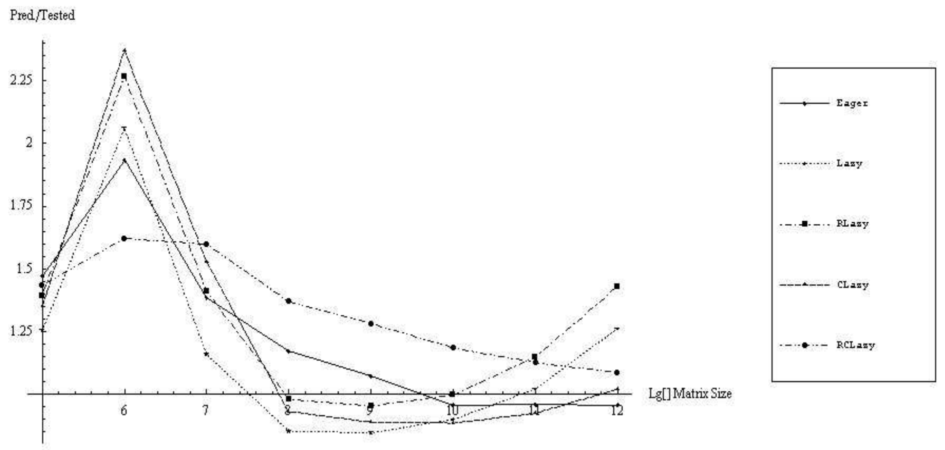


Figure 5.8: Ratio of predicted to achieved performance: 16 node Cray T3E

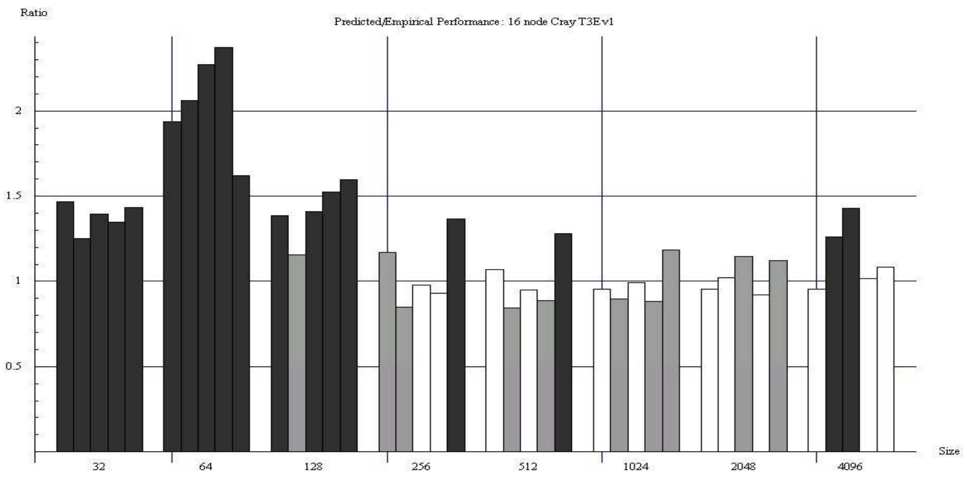


Figure 5.9: Bar graph indicating ratio of predicted to achieved performance for 16 node Cray T3E. From left-to-right the bars correspond to the Eager, Lazy, Row Lazy, Column Lazy, and Row-Column Lazy implementations.

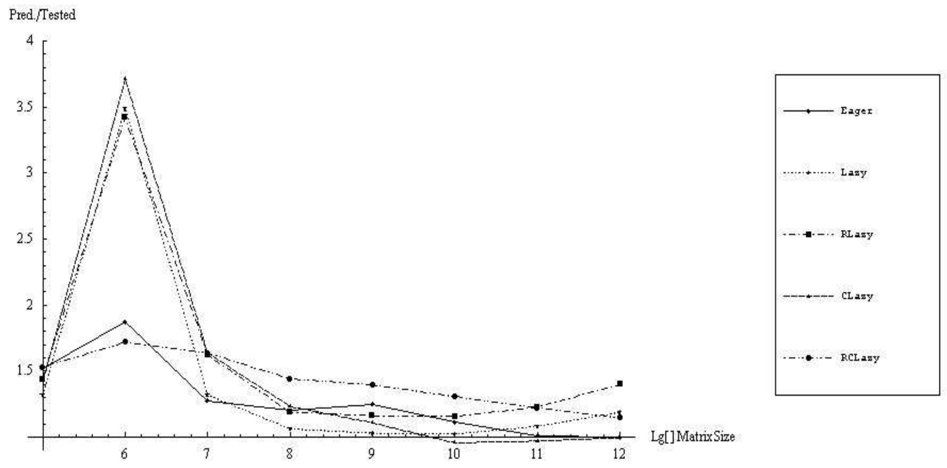


Figure 5.10: Ratio of predicted to achieved performance: 64 node Cray T3E

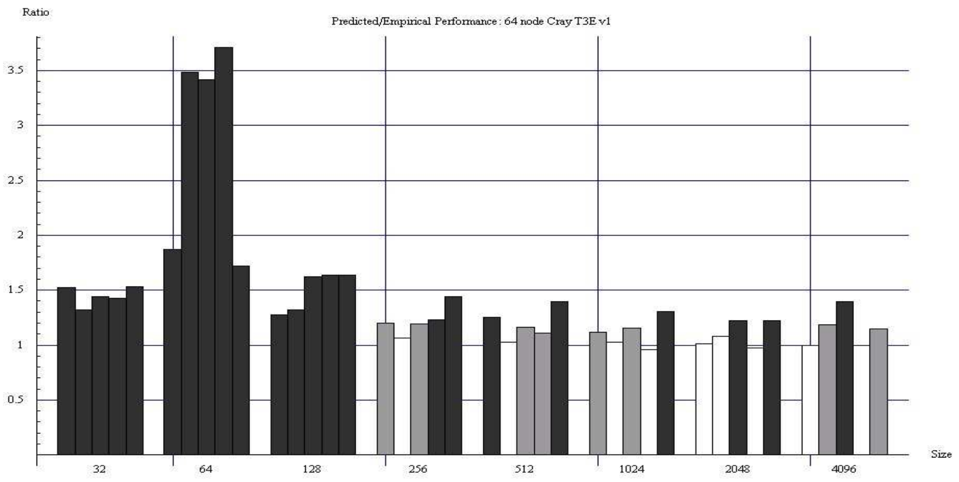


Figure 5.11: Bar graph indicating ratio of predicted to achieved performance for 64 node Cray T3E. From left-to-right the bars correspond to the Eager, Lazy, Row Lazy, Column Lazy, and Row-Column Lazy implementations.

Now, we review the graphs corresponding to the ratios of predicted/achieved performance for the four building-block algorithms (eager1, eager2, eager4, and eager5) and examine the same information regarding those routines that utilize these components (eager3a, eager3b, eager6a, eager6b, and eager6c).

Figure 5.12, Figure 5.13, and Figure 5.14) indicate the performance of the building-blocks described in Section 4.4, while Figure 5.15, Figure 5.16, and Figure 5.17 utilize these building blocks as their subcomponent LU factorization.

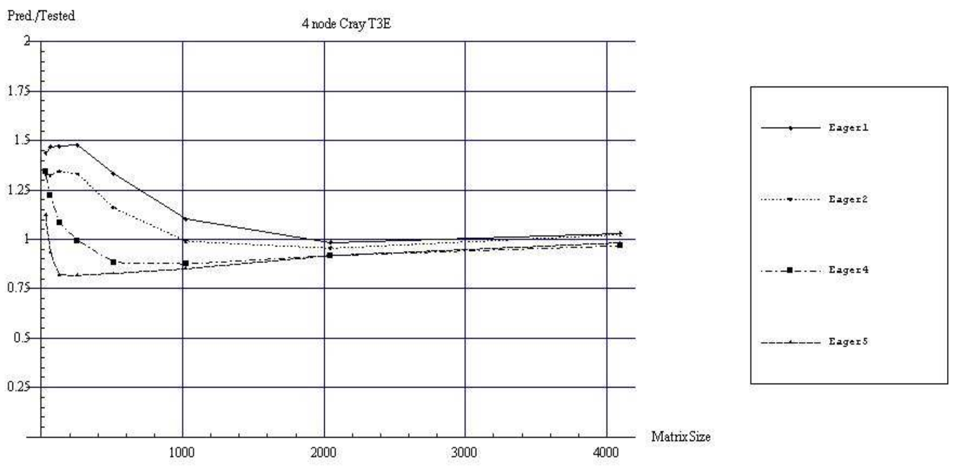


Figure 5.12: Building blocks algorithms. Ratio of predicted to achieved performance: 4 node Cray T3E

5.6.3 Experiments: A Summary

The studies in this chapter were intended to demonstrate the utility of FLAME as a method in the context of the entire environment. While Section 2.8.3 gave evidence that supported FLAME's usefulness as both a practical and pedagogical tool, the results given here are intended to lend support to the idea that much of the FLAME method can be automated and that such mechanization would prove useful.

This chapter also supplied evidence supporting the soundness of the concepts behind the PLANALYZER. The automated part of the system proved capable of:

1. Creating many code instances from the same script input.
2. Generating code instances that utilized hand-made script specializations.
3. Accurately determining the performance characteristics of a number of code instantiations.

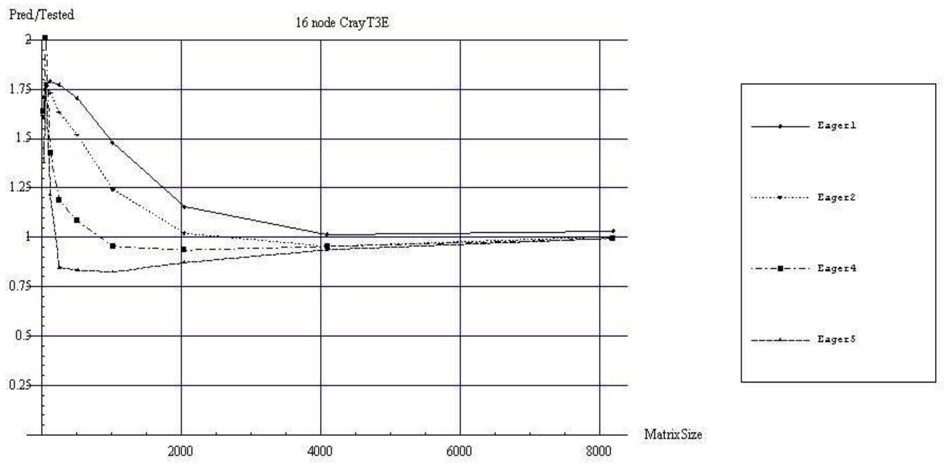


Figure 5.13: Building blocks algorithms. Ratio of predicted to achieved performance: 16 node Cray T3E

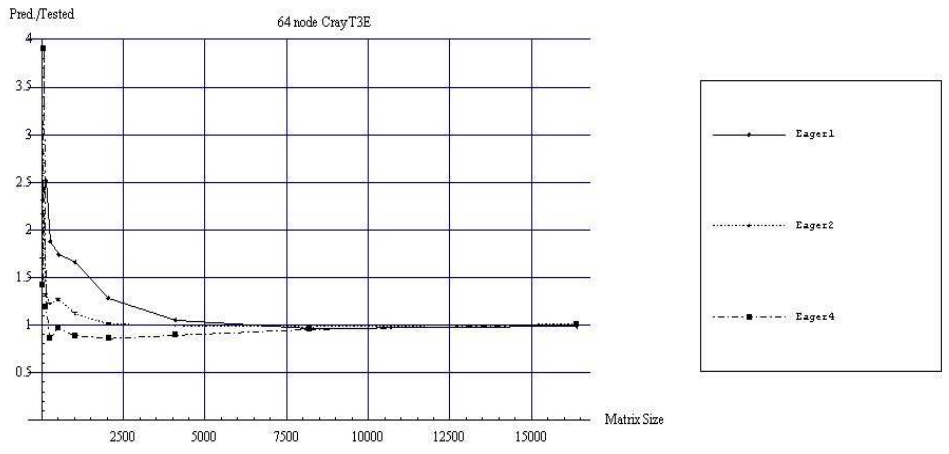


Figure 5.14: Building blocks algorithms. Ratio of predicted to achieved performance: 64 node Cray T3E

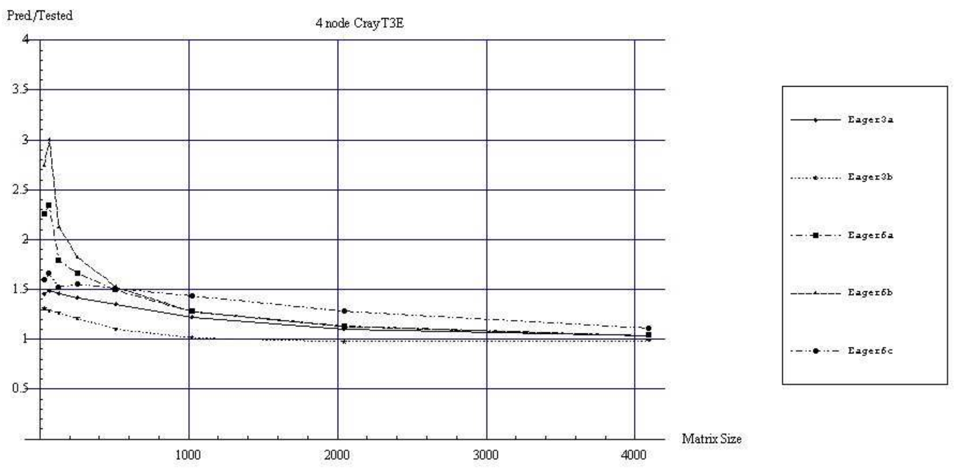


Figure 5.15: Algorithms utilizing building blocks. Ratio of predicted to achieved performance: 4 node Cray T3E

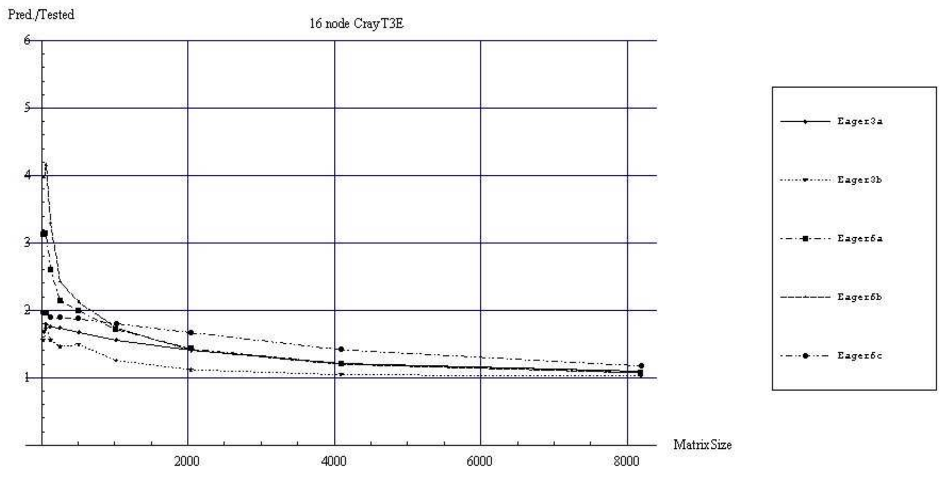


Figure 5.16: Algorithms utilizing building blocks. Ratio of predicted to achieved performance: 16 node Cray T3E

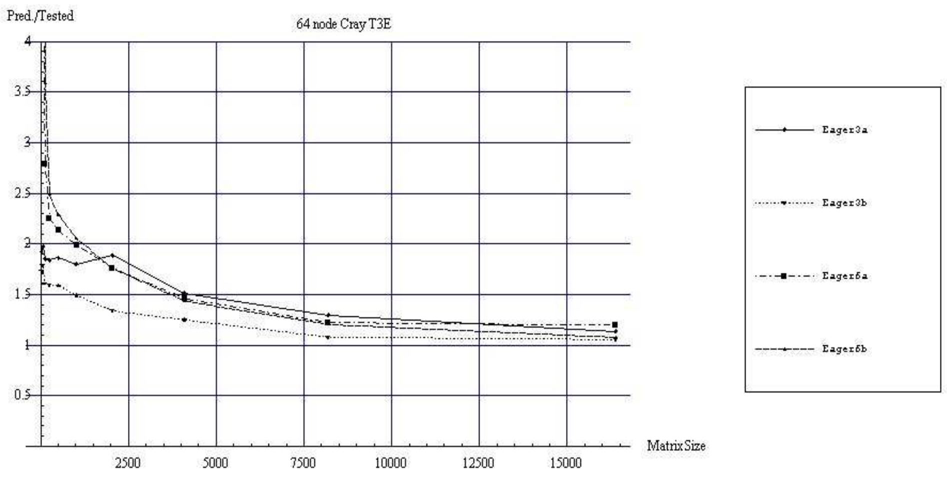


Figure 5.17: Algorithms utilizing building blocks. Ratio of predicted to achieved performance: 64 node Cray T3E

4. Using this information to hybridize a set of variants and/or version to achieve superior performance.

As these three goals are important when one is constructing scientific libraries, we think that the prototype holds up well as a proof-of-concept. Within the PLANALYZER, the performance models could be refined to give results that are more accurate or extended to give results that are more meaningful.

5.7 Chapter Summary

As has already been discussed in this chapter, performance is usually a sought after characteristic in linear algebra codes. In the case of *library* codes, this quality is even more highly prized because performance is far more important in the case of an often-invoked routine than in the case of a routine that is executed only a few times.

Of course, while the typical measure of performance is speed (i.e. the length of time the routine requires in order to execute), there are often other concerns. In some cases it is not just desirable, but *vital* to have a small memory footprint. Since there are many other axes by which "quality" can be measured, the tools should be capable of handling an assortment of metrics. Because of the details of implementation, this work can be extended to handle such things. Further, the unified nature of the development system facilitates rapid revision and specialized optimizations.

Chapter 6

Conclusion

Given a limited amount of time and/or a language that is not domain-specific and inflexible, it is often the case that one has to settle for the realization of less ambitious algorithms, little or no hybridization, one-of-each routines (i.e. a monolithic software structure), and deal with problems in cross-platform transportability.

There are time and financial penalties involved when one utilizes inefficient code. Often, there is a potential trade-off; greater resources can be devoted to a problem in order to bolster the shortcomings of the computational system. These can be in the form of human or machine resources. However, trade-offs are sometimes unavailable and are often costly.

In this dissertation, we present evidence that it is possible to create a development system that helps one in dealing with these problems. In this chapter, we present, by topic, the problems addressed and lessened by the approach and implementation described here.

6.1 Design: FLAME

While it is the only step in the development process that is not automated, the design phase is the core of the system. By deriving algorithms in a systematic manner and expressing them in a regimented form, we have the basis for automating the rest of the system. This methodology and the relatively uniform nature of the resultant algorithmic depictions facilitate the generation of multiple routines with the same functionality and, therefore, an easier path to such things as algorithmic hybridization.

Similarly, targeting specific levels of a computational system by applying *small* modifications to a uniform approach allows for vertical integration. It facilitates analysis since similar annotations are applicable to similar routines throughout the hierarchy. Example areas where this methodology has shown its efficacy range from the bottom of the memory pyramid with ITXGEMM, through PFLAMBE, to the top, POOCLAPACK, a parallel, distributed, out-of-core library.

One example of an area where FLAME might prove useful in the future involves the use of recursive data structures for storing matrices [48, 4, 46, 49]. By storing matrices by blocks rather than row- or column-major ordering, data reuse in caches can be enhanced. By combining this with recursive algorithms that exploit this data structure, impressive performance improvements have been demonstrated. Recently, work at IBM's T.J. Watson research center and The University of Umea have shown the utility of a specific type of hierarchical descriptor/storage format for matrices, namely recursive structures that go hand-in-hand with recursive algorithms [29, 46].

The crux of the design philosophy, as it relates to performance is that there are two important characteristics of modern, parallel computers: computation and communication. Virtually, all performance gains occur in the optimization of a computational or a communication routine when we view things "in the small." In a library built upon routines that ultimately rely on a very small matrix-matrix multiply kernel, virtually all of the speed-up stems from careful memory subsystem management. When considering an out-of-core library, there are more layers of memory to manage and the FLAME philosophy has been a great aid in the construction of such libraries.

Many aspects of the derivational approach we have described are systematic: the generation of the loop-invariants, the derivation of the algorithm as well as the translation to code. However, while we have much evidence to suggest that mechanizing the process is achievable, there is much work ahead.

We have demonstrated that the system presented in this document fulfills its potential by discussing how the technique has been applied to different computational environments.

6.2 Language: PLAWright

The end-user, using FLAME, should be able to encode algorithms rapidly, while introducing few errors. Both of these issues are addressed by having a programming language that is syntactically similar to the language of design. If the designer and the programmer are one and the same, this "proximity" is useful because it minimizes the possibility of a mistranslation between the two forms of the algorithm. If the implementor and designer are two distinct entities, this resemblance of form has an additional advantage, namely, lessening the likelihood of a misinterpretation of the design before it is translated into input for the system.

Encapsulated, the benefits of the PLAWright programming language are:

1. It closely resembles the language of the algorithms.
2. It can be written at a very high level or at a lower level.
3. The transition from general to specialized is both smooth and flexible.
4. It can be described using typical compiler formalizations.

6.3 Automated Code Generation: PLANalyzer

Typically, the construction of a linear algebra library requires the implementation of a large number of algorithms. The derivation process advanced in this work is applicable to those algorithms at the core of dense linear algebra, exhibits a systematic nature that lends itself to rapid derivation, and produces algorithms in a form that can be mechanically translated into input for the PLANALYZER code production tool. Similarly, the code manufacturing system can address the same spectrum of algorithms as the derivation system, is mechanical, and is relatively fast. Therefore, algorithmic coverage can be quickly achieved by one familiar with the derivation methodology.

In the best of all possible worlds, the automatically generated code would also be *provably correct*. Given the formal approach provided by FLAME and the nature of the code generation facilities presented in Chapter 4 of this dissertation, we think that this is possible for the domain-specific language presented in Chapter 3.

A domain-specific language provides a set of high-level operations that are convenient for a specific domain. If we formalize the syntax and semantics of a domain-specific language, then we can use formal methods to prove that a program written in a domain-specific language is correct. That the implementation of the domain-specific language is correct is an orthogonal issue, related to low-level compiler verification, and ably handled by others.

Towards this end, a collaborative effort with Dr. Panagiotis Manolios targets the following:

1. Proving that for *any* PLAWright code, the PLANALYZER's output is a legal PLAPACK program with the same semantics as that of the input script.

2. Applying tactic-based theorem proving to construct a system that utilizes both the input and output of the integrated PLANALYZER system and, on a per instance basis, creates proofs of correctness.

6.4 Automated Analysis: plANALYZER

Performance is one of the paramount concerns in the area of linear algebra library construction. There are three interrelated facets of this issue that need to be dealt with: modeling the environment, evaluating the performance estimates, and using the result of the evaluation. All three issues have been dealt with by the system described in this dissertation.

It is rarely the case that the code that achieves optimal performance on one architecture will perform as admirably on another. It is therefore a common goal to have code that is performance portable across various systems. The work presented here includes the use of a high-level language in conjunction with analysis technology. This facilitates the production of performance transportable code.

6.5 An Integrated System: FLAME and PLANALYZER

To construct a linear algebra library one must design and implement the algorithms that must be available to the library user and make them as efficient as possible. FLAME provides a systematic means for deriving the variants of such algorithms. The PLAWright compiler allows for rapid prototyping. Automatic generation of the code corresponding to the PLAWright script is handled by the compiler (PLAN) component of the PLANALYZER. Finally, the analytical (ANALYZER) component of the system yields information regarding the performance characteristics of the produced code, opening the door for hybridization.

We have explored the development of all of the concepts and tools necessary for a methodical hybridization of a linear algebra library and believe that we have made a strong case for the soundness of the approach presented in this dissertation.

Bibliography

[1] R.C. Agarwal, F.G. Gustavson, and M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, 38(5), Sept. 1994.

[2] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J. Wu. PLAPACK: Parallel linear algebra package – design overview. In *Proceedings of SC97*, 1997.

[3] B. Alpern and L. Carter. Performance programming: A science waiting to happen, 1994.

[4] Bjarne S. Andersen, Fred G. Gustavson, and Jerzy Wasniewski. A recursive formalation of Cholesky factorization of a matrix in packed storage. LAPACK Working Note 146 CS-00-441, University of Tennessee, Knoxville, May 2000.

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

[6] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. Basic Linear Algebra Communication Subprograms. In *Sixth Distributed Memory Computing Conference Proceedings*, pages 287–290. IEEE Computer Society Press, 1991.

[7] Greg Baker, John Gunnels, Greg Morrow, Beatrice Riviere, and Robert van de Geijn. PLAPACK: High performance through high level abstraction. In *Proceedings of ICCP98*, 1998.

[8] S. Balay, W. Gropp, L. McInnes, and B. Smith. Efficient management of parallelism in object oriented numerical software libraries, 1997.

[9] Satish Balay, William Gropp, Lois Curfman McInnes, and Barry Smith. PETSc 2.0 users manual. Technical Report ANL-95/11, Argonne National Laboratory, Oct. 1996.

[10] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, , and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1996.

[11] J. Bilmes, K. Asanovic, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*. ACM SIGARC, July 1997.

[12] Walter R. Bischofberger. Sniff: A pragmatic approach to a c++ programming environment. In *C++ Conference*, pages 67–82, 1992.

[13] L. S. Blackford, A. Cleary, J. Demmel, J. Dongarra, I. Dhillon, S. Hammarling, A. Petitet, H. Ren, K. Stanley, , and R. C. Whaley. Practical experience in the dangers of heterogeneous computing. *ACM Trans. Math. Soft.*, to appear.

[14] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. LAPACK Working Note 100 CS-95-292, University of Tennessee, May 1995.

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

[16] Gene Cooperman. STAR/MPI: Binding a parallel library to interactive symbolic algebra systems. In *International Symposium on Symbolic and Algebraic Computation*, pages 126–132, 1995.

[17] Bimillennium Corporation. Hiq reference manual, version 2.0, 1993.

[18] P. D. Crout. A short method for evaluating determinants and solving systmes of linear equations with real or complex coefficients. *Trans AIEE*, 60:1235–1240, 1941.

[19] L. DeRose and D. Padua. A matlab to fortran 90 translator and its effectiveness. In *Proceedings of the 10th ACM International Conference on Supercomputing*, 1996.

[20] Luiz Antonio DeRose. *Compiler Techniques for MATLAB Program*. PhD thesis, Computer Sciences Department, The University of Illinios at Urbana–Champaign, 1996.

[21] Edsger Wybe Dijkstra. Under the spell of Leibniz's dream. Technical Report EWD1298, The University of Texas at Austin, April 2000. `http://www.cs.utexas.edu/users/EWD/`.

[22] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, 1979.

[23] J. J. Dongarra, F. G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, Jan. 1984.

[24] Jack Dongarra, Robert van de Geijn, and David Walker. Scalability issues affecting the design of a dense linear algebra library. *J. Parallel Distrib. Comput.*, 22(3), Sept. 1994.

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

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

[27] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* SIAM, Philadelphia, PA, 1991.

[28] C. Edwards, P. Geng, A. Patra, and R. van de Geijn. Parallel matrix distributions: have we been doing it all wrong? Technical Report TR-95-40, Department of Computer Sciences, The University of Texas at Austin, 1995.

[29] E. Elmroth and F.G. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM J. Res. Develop.*, 44(4):605–624, 2000.

[30] Kathi Fisler, Shriram Krishnamurthi, and Don Batory. Verifying component-based collaboration designs. In *ICSE Workshop on Component-Based Software Engineering*, page to appear, May 2001.

[31] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume I. Prentice Hall, 1988.

[32] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, 1990.

[33] Gene Golub and James M. Ortega. *Scientific Computing: an Introduction with Parallel Computing.* Academic Press, 1993.

[34] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, 2nd edition, 1989.

[35] John W. Gray. *Mastering Mathematica Programming Methods and Applications.* Academic Press, 2nd edition, 1997.

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

[37] William Gropp. An introduction to performance debugging for parallel computers. Technical report.

[38] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, 1996.

[39] J. A. Gunnels, D. S. Katz, E. S. Quintana-Orti, , and R. A. van de Geijn. Fault-tolerant high-performance matrix multiplication: Theory and practice. In *International Conference on Dependable Systems and Networks*, 2001.

[40] John Gunnels, Calvin Lin, Greg Morrow, and Robert van de Geijn. 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)*, pages 110–116, 1998.

[41] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. Formal Linear Algebra Methods Environment (FLAME): Overview. FLAME Working Note #1 CS-TR-00-28, Department of Computer Sciences, The University of Texas at Austin, Nov. 2000.

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

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

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

[45] Brian C. Gunter, Wesley C. Reiley, and Robert A. van de Geijn. 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, 2001.

[46] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. In B. Kågström et al., editor, *Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, Lecture Notes in Computer Science 1541, pages 195–206. Springer-Verlag, 1998.

[47] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Superscalar GEMM-based level 3 BLAS – the on-going evolution of a portable and high-performance library. In B. Kågström et al., editor, *Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, Lecture Notes in Computer Science 1541, pages 207–215. Springer-Verlag, 1998.

[48] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[49] F.G. Gustavson and I. Jonsson. Minimal storage high-performance Cholesky factorization via blocking and recursion. *IBM J. Res. Develop.*, 44(6):823–850, November 2000.

[50] S. Guyer and C. Lin. Broadway: A software architecture for scientic computing, 2000.

[51] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Domain-Specific Languages*, pages 39–52, 1999.

[52] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *TOMS*, 24(3):268–302, 1998.

[53] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjaming-Cummings, 1994.

[54] William Jalby Kyle A. Gallivan, Bret A. Marsolf and Ahmed H. Sameh. On the development of libraries and use in applications. CSRD Report 1341, Center for Supercomputing Research and Development, University of Illinois, May 1995.

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

[56] J. Li, A. Skjellum, and R. D. Falgout. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Concurrency: Practice and Experience*, 9(5):345–389, 1997.

[57] Bret Andrew Marsolf. *Techniques for the Interactive Development of Numerical Linear Algebra Libraries for Scientific Computation.* PhD thesis, Computer Sciences Department, University of Illinois at Urbana–Champaign, 1997.

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

[59] Greg Morrow and Robert van de Geijn. A parallel linear algebra server for matlab-like environments. In *Proceedings of SC98*, to appear.

[60] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A portable "shared-memory" programming model for distributed memory computers. pages 340–349, 1994.

[61] Terence Parr. Reference guide: Pccts and c++.

[62] Terence John Parr. An overview of SORCERER: A simple tree-parser generator. Technical report, 1994.

[63] M. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. Perliminary results from a matlab compiler. In *Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98)*, pages 81–87, 1998.

[64] Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. Gauss-jordan based matrix inversion and its parallelization. *SJSC*, submitted.

[65] Enrique S. Quintana-Ortí and Robert van de Geijn. Fast parallel kernels for selected problems in control theory. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[66] Wesley C. Reiley. Efficient parallel out-of-core implementation of the Cholesky factorization. Technical Report CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin, Dec. 1999. Undergraduate Honors Thesis.

[67] Wesley C. Reiley and Robert A. van de Geijn. POOCLAPACK: Parallel Out-of-Core Linear Algebra Package. Technical Report CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin, Nov. 1999.

[68] B. T. Smith et al. *Matrix Eigensystem Routines – EISPACK Guide*. Lecture Notes in Computer Science 6. Springer-Verlag, New York, second edition, 1976.

[69] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[70] S. Stanley, Kendall. Execution time of symmetric eigensolvers. Technical Report CSD-99-1039, 3, 1999.

[71] G. W. Stewart. *Matrix Algorithms Volume 1: Basic Decompositions*. SIAM, 1998.

[72] Anne E. Trephethen, Vijay S. Menon, Chi-Chao Chang, Grezgorz J. Czajkowki, Chris Myers, and Lloyd N. Trefethen. Multimatlab: Matlab on multiple processors. Technical Report 96–239, Cornell Theory Center, 1996.

[73] Robert van de Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, to appear.

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

[75] George Karypis Vipin Kumar and Ananth Grama. Role of message-passing in performance oriented parallel programming. In *Proceedings of the Eighth SIAM Parallel Processing Conference*, 1997.

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

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

Vita

John A. Gunnels was born in Long Beach, California on May 15, 1965, the son of Willis Aaron and Margaret Mary Gunnels. He attended Redmond High School, Central Oregon Community College, Oregon State University, and The University of Illinois, Urbana-Champaign before moving to Austin where he met Jen Moore at UT Austin in 2001. He currently resides in Mt. Kisco, NY with his dog, Data.

Permanent Address: 85 Foxwood Circle
Mt. Kisco, NY 10549

This dissertation was set by the author using the LaTeX 2_ε typesetting system.