Parallel Implementation of Triangular Solve

Thierry Joffrain

May 12, 1998

CS379H Honors Thesis:	Parallel Implementation of Triangular Solve
Supervising Professor:	Robert A. van de Geijn

Abstract

Triangular solve algorithms are derived for both sequential and parallel architectures. Several variants are analyzed and compared from the similarities they exhibit. This leads to a detailed discussion of various approaches toward the implementation of high performance algorithms. A single multi-faceted high performance algorithm is then derived. Implementation using PLAPACK and resulting performance is reported.

Contents

Chapter 1

Introduction

The solution of a system of linear equations with n equations and n unknowns is generally presented in matrix form as $Ax = b$, where A is an $n \times n$ matrix and both x and b are vectors of length n. While this general system of equations requires $O(n^3)$ operations, the special case where A is a triangular matrix (that is, has only zeroes above or below the main diagonal) is considerably cheaper to solve, requiring $O(n^2)$ operations[1]. In this paper, we deal with the parallel implementation of this case, which is generally referred to as a triangular solve.

As the triangular solve is most often used in conjunction with other arithmetic operations, which are themselves $O(n^3)$, its performance is often abandoned as a secondary concern. This usually results in a less than efficient implementation which becomes a costly problem when the triangular solve is used repeatedly.

Due to the nature of the triangular solve, there is a dependency between computational steps. In the mid 1980's, this led Cleve Moler to declare there to be no parallelism in the triangular solve [5]. This statement generated several papers, all published in the same issue of the SIAM Journal on Scientific and Statistical Computing, which proved him wrong [1, 4, 6].

In particular, three classes of algorithms were proposed: "fan-in, fan-out" algorithms, "wavefront" algorithms, and "cyclic" algorithms [1, 13]. Fan-in fan-out and wavefront algorithms are related. They also lead to cyclic algorithms, often referred to as Li-Coleman algorithms [3], which were first proposed by Lennart Johnsson [13], a fact that is often overlooked. Since these papers were written, there have been no publications of significance regarding this subject.

1.1 Background

In 1988, Heath and Romine [1] discussed and named two serial algorithms. When executed on the average personal computer, these are an adequate approach to the triangular solve problem $Lx = b$. Figures (1.1) and (1.2) represent their algorithms, β and χ representing vectors elements, λ a matrix elements. They use doubly nested loops which can be interchanged. This, they pointed out, leads to a duality in the algorithms. The names they chose for each al-

[1] Two loops, one nested in the other, as will be shown later in this chapter

gorithm ("Scalar-Product" and "Vector-Sum") re-emphasize the duality and characterize the computation done in their innermost loop.

$$
\begin{array}{l}
\textbf{for } i = 1 \textbf{ to } n \\
\quad \textbf{for } j = 1 \textbf{ to } i - 1 \\
\quad\quad \beta_i = \beta_i - \chi_j \lambda_{ij} \\
\quad \chi_i = \beta_i / \lambda_{ii}
\end{array}
$$

Figure 1.1: Scalar-Product serial algorithm

$$
\begin{array}{l}
\textbf{for } j = 1 \textbf{ to } n \\
\quad \chi_j = \beta_j / \lambda_{jj} \\
\quad \textbf{for } i = j + 1 \textbf{ to } n \\
\quad\quad \beta_i = \beta_i - \chi_j \lambda_{ij}
\end{array}
$$

Figure 1.2: Vector-Sum serial algorithm

Given the nature of the computation, a more adequate name for the scalar-product and vector-sum algorithms would be "lazy algorithm" and "greedy algorithm" respectively. This will be motivated further in later chapters.

1.1.1 Distributed Memory Parallel Computing

Parallel computers consist of a collection of processors, also known as nodes. There are two major memory paradigms in parallel computers. One is shared memory, where memory is one large common pool accessed by all processors. Communication between nodes is then usually done through this memory. The other paradigm is distributed memory, where each node has it's own memory, with all nodes connected in a certain network topology, or mesh, for communication. This mesh is made up of a number of rows and columns of nodes.

The direct effect of distributed memory is that data is distributed among nodes, all of it rarely residing completely on only one at any time. This is especially true when the problem is such that it cannot fit on a single node.

Both the IBM SP2 and the Cray T3E, the machines used in this thesis, follow the distributed memory model.

1.1.2 Data Distribution

In a distributed memory environment, there are a number of algorithm for distributing data, Wrapping data on the node mesh – which involves putting data on each node one after the other until all the data is distributed – is frequently used. A one-dimensional wrapping would involve storing columns or rows of data along columns or rows of processors, looping around the processors when there is more data. A two-dimensional wrapping wraps data in both column and row directions.

The distribution of the data is crucial to the computation. Clusters of data which need to be manipulated together are required to exist on the same node for the computation to take place. Hence the data should be distributed in a way consistent with that of the computation. Such a distribution, where the layout of parts of an object are consistent with where an application would need them, is called *physically based* distribution.

1.1.3 Basic Linear Algebra Subprograms

Basic Linear Algebra Subprograms[8, 9, 10], generally referred to as BLAS, are computational kernels widely used by applications and libraries dealing with linear algebra. The BLAS routines perform linear algebra operations such as inner product, matrix-vector and matrix-matrix multiplication. Since these kernels are standard, vendors can develop highly optimized routines for their platforms, allowing high performance implementations to be portable.

The BLAS standard defines operations in three distinct categories, called levels. Level 1 deals with vector-vector operations, such as the inner (dot) product. For these operations, $O(n)$ computation is performed on $O(n)$ data.

Level 2 deals with matrix-vector operations. The single right hand side triangular solve falls in this category and for these operations, $O(n^2)$ computation is performed on $O(n^2)$ data.

Level 3 deals with matrix-matrix operations. Triangular solves with multiple right hand side falls in this category. For these operations, $O(n^3)$ computation is performed on $O(n^2)$ data.

1.1.4 MPI

With the rapid advances in technology and the fast pace of change in hardware, MPI (Message-Passing Interface)[12] was developed as a communication library for parallel computers and networks of workstations. It is the proposed standard for message passing operations in that target environment.

MPI was developed by a broad group of software writers, application scientists and parallel computer vendors as a portable library. MPI itself is a specification for a library of routines. Calls to such a library are easily made from FORTRAN or C.

1.1.5 PLAPACK

PLAPACK (Parallel Linear Algebra PACKage)[7, 11] is a library of infrastructures that allows quick translation of sequential algorithms of linear algebra operations into parallel methods without knowledge of any parallel programming paradigm. It has been created at the University of Texas at Austin by Professor Robert A. van de Geijn and his team.

This library uses MPI. As it has been written in C and makes use of the BLAS routines for computation, it has been ported to various architectures, among which the IBM SP2 and Cray T3E.

PLAPACK's data distribution is based on a two dimensional wrapping, tighter among rows than columns, which generates a physically based distribution.

1.1.6 Notation

In this paper, certain conventions are used. Scalars are represented by Greek letters, α, β, etc. Vectors are represented by lower case letters a, b, etc. All vectors are assumed to be vertical, unless marked as $\hat{a}^T$. Matrices are represented by uppercase letters A, B, etc. A lower triangular matrix is thus represented by L and an upper triangular matrix by U.

When representing vectors or matrices, double vertical or horizontal lines are used to indicate which portions of the linear algebra objects have been used. For example, at the beginning of a triangular solve, an unknown vector x may be described in the following way, to show that χ_1 and x_2 have not been computed:

$$\left(\begin{array}{c} x_0 \\ \hline\hline \chi_1 \\ \hline x_2 \end{array}\right)$$

After the computation of χ_1 then the vector is described as:

$$\left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline\hline x_2 \end{array}\right)$$

This shift in the double line essentially shows the progression of the algorithm through the object, including the direction of the progress. Although in two dimensions, the same principle applies for parts of matrices.

The tilde notation indicates a partial computational result due to an update. For example, a vector x represented as $\tilde{x}$ indicates this vector has been updated by the result of a computation, but this update is only an intermediary result.

1.2 Related Work

From the greedy and lazy algorithms presented earlier, Heath and Romine proceeded to examine three types of parallel algorithms: fan-in fan-out, cyclic and wavefront. As with the serial algorithms, we present a brief overview of each, with comments on their features and duality they might exhibit.

1.2.1 Fan-out and Fan-in Algorithms.

The fan-out fan-in algorithms[2] gain increased performance by distributing the work done in the inner loop of each algorithm between the nodes, so that the bulk of the work is shared. The name of the algorithms was coined from the pattern of the communication required. First the relevant data is distributed using a broadcast from one node to the others (fan-out) and then the results are gathered back to one node (fan-in). The fan-out is the synchronizing step.

This distribution is equivalent to a run-time partitioning of the data and performance increases as the size of the each partition increases. The broadcast and gather functions are implemented in the same way for both the greedy and lazy versions of the algorithm. In effect the existing duality is unchanged.

[2] Also known as "column-sweep" [2]

1.2.2 Wavefront Algorithms

Wavefront algorithms achieve increased performance by working on many parts of the solution at once. However, the inherent data dependencies result in strict limitations.

The first of the dual algorithms requires that the data be distributed by row, among rows of processors. When the top row has completed the very first step of computation, that is, as soon as the first entry of b has been computed, the second row of processors can start work on its row. Computation then proceeds in like fashion. It is the outer loop which is partitioned in this case. If there are fewer rows of processors than rows of data, the data is wrapped over the processor rows. Row distribution is the required distribution for the lazy approach. On the other hand for the greedy approach, a column distribution is required.

The distribution used in each results in an algorithmic requirement which must be satisfied. Furthermore, from the algorithms result a dependency on data distribution. This reduces the flexibility of this type of algorithm.

In wavefront algorithms, one row is distributed at a time. A more efficient implementation is one which uses blocks of rows or columns.

1.2.3 Cyclic Algorithms

First proposed by Lennart Johnsson [13], the algorithm developed by Li and Coleman [3, 4] falls into this category and shares the characteristic traits of these types of algorithms. Their trademark involves reduced communication volume through an explicit passing of a fixed amount of data.

In previous algorithms, when working within a row or column, it is the length of the solution vector used that grows with each step, as more of it is calculated. The cyclic algorithms instead repeatedly pass around a bucket of constant size containing the required information, hence the term cyclic. The length of this bucket is specially chosen as $p-1$, where p is the number of processors, so that the circulation of this bucket does not produce overlaps. The bucket is used to accumulate partial local results and distributing them.

1.3 Acknowledgments

It is the work of the PLAPACK group which has made this thesis possible and meaningful. I am particularly grateful to Professor Robert van de Geijn, heading this group and from whom I was fortunate to take a class, for his advice, time and inspiration when tackling new problems. I would like to thank Greg Morrow and John Gunnels also, for their insight, as well as for answering questions that allowed me to take a look under the hood of PLAPACK.

Chapter 2

Sequential Triangular Solve

2.1 Identifying The Triangular Solve

The single-side triangular solve is defined in general as

$$Ax = b \text{ or } A^T x = b$$

where A is an upper or lower triangular matrix with dimensions $n \times n$, and both x and b are vectors.

2.1.1 Classification of Cases and Variants

	trans	
upper, lower	no transpose	transpose
lower triangular	$Lx = b$	$L^T x = b$
upper triangular	$Ux = b$	$U^T x = b$

Each of these four cases can in turn be computed through either of two variants: lazy or greedy algorithms.

We begin with an outline of the general properties common to all of the algorithms which will be discussed in this thesis. Later we introduce and deal with particular examples of the dual algorithms at the core of the problem. The lazy algorithm presented exemplifies the properties of other lazy algorithms, and is used to illustrate each detail of these. Following this, these details will be assumed and mentioned only when required for clarity. The same applies to the subsequent greedy algorithm.

In the next chapters, the lower triangular, no transpose case is used throughout. While all four cases could be discussed in detail at each step, we will show that they differ only in the progression through the objects during the computation, not in the computation itself.

2.1.2 General Properties of Algorithms

Consider the lower triangular solve $Lx = b$, partitioned in the following way:

$$L = \left(\begin{array}{c||c} L_{TL}^{(k)} & 0 \\ \hline\hline L_{BL}^{(k)} & L_{BR}^{(k)} \end{array} \right), \quad x = \left(\begin{array}{c} x_T^{(k)} \\ \hline\hline x_B^{(k)} \end{array} \right), \quad \text{and} \quad b = \left(\begin{array}{c} b_T^{(k)} \\ \hline\hline b_B^{(k)} \end{array} \right)$$

The superscript indicates the step number in the computation. TL, BR, etc, indicate a particular portion in relation to the whole. Due to the nature of the partitioning, it is a property of both $L_{TL}^{(k)}$ and $L_{BR}^{(k)}$ to be themselves proper lower triangular, the former being $k \times k$ and the latter $(n-k) \times (n-k)$. This means that $x_T^{(k)}$ and $b_T^{(k)}$ are both of length k.

Then

$$Lx = \left(\begin{array}{c} L_{TL}^{(k)} x_T^{(k)} \\ \hline\hline L_{BL}^{(k)} x_T^{(k)} + L_{BR}^{(k)} x_B^{(k)} \end{array} \right) = \left(\begin{array}{c} b_T^{(k)} \\ \hline\hline b_B^{(k)} \end{array} \right)$$

yielding the equalities

$$\begin{aligned} L_{TL}^{(k)} x_T^{(k)} &= b_T^{(k)} && (2.1) \\ L_{BL}^{(k)} x_T^{(k)} + L_{BR}^{(k)} x_B^{(k)} &= b_B^{(k)} && (2.2) \end{aligned}$$

In both the greedy and lazy algorithms presented below, it is assumed that at step k, $L_{TL}^{(k)} x_T^{(k)} = b_T^{(k)}$ has been solved. This leaves a smaller but similar problem of the form $L_{BR} x_B^{(k)} = b_B^{(k)} - L_{BL}^{(k)} x_T^{(k)}$ to be solved.

The algorithms differ by what other computation has been performed in addition to that of $x_T^{(k)}$. Parts of $b_B^{(k)}$ may or may not have been updated, depending on the algorithm. This determines the nature of the algorithm and the assumptions they depend on. Solving the smaller subproblem can be regarded as a recursive step, termination of the whole problem being determined when that subproblem has size zero.

2.2 Sequential Implementation: No More Than Simple

The simplest implementation of the single side triangular solve makes no attempt to optimize the computation beyond the sequential architecture for which it was designed. Beyond what support can be provided by a good parallel code compiler, porting to a parallel architecture yields less than acceptable results. In fact, achieving 5 to 15 percent (with a *good* compiler!) of machine peak is a standard result.

One such standard algorithm is:

$$\begin{array}{l} \textbf{for } i = 1 \textbf{ to } n \\ \quad \textbf{for } j = 1 \textbf{ to } i-1 \\ \quad\quad \beta_i = \beta_i - \chi_j \lambda_{ij} \\ \quad \chi_i = \beta_i / \lambda_{ii} \end{array}$$

Figure 2.1: Fig 1. Lazy serial algorithm

The partitioning is

$$L = \left(\begin{array}{c|c|c} \lambda_{11} & \cdots & 0 \\ \hline \vdots & \ddots & \vdots \\ \hline \lambda_{n1} & \cdots & \lambda_{nn} \end{array} \right), \quad x = \left(\begin{array}{c} \chi_1 \\ \hline \vdots \\ \hline \chi_n \end{array} \right), \quad \text{and} \quad b = \left(\begin{array}{c} \beta_1 \\ \hline \vdots \\ \hline \beta_n \end{array} \right)$$

$$
\begin{array}{l}
\textbf{for } j = 1 \textbf{ to } n \\
\quad \chi_j = \beta_j / \lambda_{jj} \\
\quad \textbf{for } i = j+1 \textbf{ to } n \\
\quad\quad \beta_i = \beta_i - \chi_j \lambda_{ij}
\end{array}
$$

Figure 2.2: Fig 2. Greedy serial algorithm

2.2.1 Lazy algorithm

A lazy algorithm is one which will, at each step, do the minimum amount of computation required for the next step to proceed. By the same token, in a recursive setting, it also implies that the lazy algorithm makes the minimal amount of assumptions with regards to what the previous step computed.

For the single side, no-transpose lower triangular solve, the lazy algorithm assumes that *only* the subproblem consisting of $x_T^{(k)}$ (2.1) has been solved. This is the minimum knowledge required to carry out step $k+1$. The question now becomes

> What computation must be performed so that $x_T^{(k+1)}$ is computed, where
>
> $$L_{TL}^{(k+1)} x_T^{(k+1)} = b_T^{(k+1)} \tag{2.3}$$

Repartition L, x and b:

$$
L = \left(\begin{array}{c||c|c} L_{00}^{(k)} & 0 & 0 \\ \hline\hline l_{10}^{(k)\,T} & \lambda_{11}^{(k)} & 0 \\ \hline L_{20}^{(k)} & l_{21}^{(k)} & L_{22}^{(k)} \end{array} \right), \quad x = \left(\begin{array}{c} x_0^{(k)} \\ \hline\hline \chi_1^{(k)} \\ \hline x_2^{(k)} \end{array} \right), \quad \text{and} \quad b = \left(\begin{array}{c} b_0^{(k)} \\ \hline\hline \beta_1^{(k)} \\ \hline b_2^{(k)} \end{array} \right)
$$

where $L_{00} = L_{TL}$ and the other quadrants delimited by the double lines match the rest of our generic partition described earlier. Likewise for the x and b vectors, where $x_T^{(k)} = x_0^{(k)}$ and $b_T^{(k)} = b_0^{(k)}$. The focus of this step is to compute $x_T^{(k+1)}$. Since $x_0^{(k)}$ is known, this is done by solving for $\chi_1^{(k)}$.

Then

$$
Lx = \left(\begin{array}{c} L_{00}^{(k)} x_0^{(k)} \\ \hline\hline l_{10}^{(k)\,T} x_0^{(k)} + \lambda_{11}^{(k)} \chi_1^{(k)} \\ \hline L_{20}^{(k)} x_0^{(k)} + l_{21}^{(k)} \chi_1^{(k)} + L_{22}^{(k)} x_2^{(k)} \end{array} \right) = \left(\begin{array}{c} b_0^{(k)} \\ \hline\hline \beta_1^{(k)} \\ \hline b_2^{(k)} \end{array} \right)
$$

yielding the equalities

$$
\begin{aligned}
L_{00}^{(k)} x_0^{(k)} &= b_0^{(k)} && (2.4) \\
l_{10}^{(k)\,T} x_0^{(k)} + \lambda_{11}^{(k)} \chi_1^{(k)} &= \beta_1^{(k)} && (2.5) \\
L_{20}^{(k)} x_0^{(k)} + L_{21}^{(k)} \chi_1^{(k)} + L_{22}^{(k)} x_2^{(k)} &= b_2^{(k)} && (2.6)
\end{aligned}
$$

Since $x_0^{(k)} = x_T^{(k)}$ and

$$
x_T^{(k+1)} = \left(\begin{array}{c} x_0^{(k)} \\ \hline\hline \chi_1^{(k)} \end{array} \right)
$$

it suffices to compute the following:

$$\tilde{\beta}_1^{(k)} = \beta_1^{(k)} - l_{10}^{(k)\,T} x_0^{(k)} \tag{2.7}$$
$$\chi_1^{(k)} = \tilde{\beta}_1^{(k)} / \lambda_{11}^{(k)} \tag{2.8}$$

Since we have the required answer, $\chi_1^{(k)}$, nothing else needs to be done and the next step can be initiated.

We note that b_1 does not occur in the third of our equalities (2.6). It is thus safe to overwrite b_1 with x_1. Indeed, this means that vectors b and x need only be symbolically different and can actually be physically identical, saving storage space.

At the end of this step, we have the following:

$$L = \left(\begin{array}{c||c} L_{TL}^{(k+1)\,T} & 0 \\ \hline\hline L_{BL}^{(k+1)} & L_{BR}^{(k+1)} \end{array}\right) = \left(\begin{array}{c|c||c} L_{00}^{(k)} & 0 & 0 \\ \hline l_{10}^{(k)\,T} & \lambda_{11}^{(k)} & 0 \\ \hline\hline L_{20}^{(k)} & l_{21}^{(k)} & L_{22}^{(k)} \end{array}\right),$$

$$x = \left(\begin{array}{c} x_T^{(k+1)} \\ \hline\hline x_B^{(k+1)} \end{array}\right) = \left(\begin{array}{c} x_0^{(k)} \\ \hline \chi_1^{(k)} \\ \hline\hline x_2^{(k)} \end{array}\right), \quad \text{and} \quad b = \left(\begin{array}{c} b_T^{(k+1)} \\ \hline\hline b_B^{(k+1)} \end{array}\right) = \left(\begin{array}{c} b_0^{(k)} \\ \hline \beta_1^{(k)} \\ \hline\hline b_2^{(k)} \end{array}\right)$$

2.2.2 Greedy algorithm

A greedy algorithm is one which will, at each step, do the maximum amount of computation it can complete, before the next step proceeds. By the same token, in a recursive setting, it also implies that the greedy algorithm makes the maximum amount of assumptions with regards to what the previous step had computed.

For the single side, no transpose lower triangular solve, the greedy algorithms assumes that the subproblem $x_T^{(k)}$ (2.1) has been solved *and* that the rest of the vector b has been updated with the partial results from previously completed steps. Not only $x^{(k)}$ but also $b_B^{(k)} \leftarrow b_B^{(k)} - L_{BL}^{(k)} x_T^{(k)}$ have been computed. Although different, the question is similar and becomes

> What computation must be performed so that $x_T^{(k+1)}$ is computed, where
>
> $$\tilde{b}_B^{(k+1)} = b_B^{(k+1)} - L_{BL}^{(k+1)} x_T^{(k+1)} \tag{2.9}$$
> $$L_{BR}^{(k+1)} x_T^{(k+1)} = \tilde{b}_T^{(k+1)} \tag{2.10}$$

Repartition L, x and b.

$$L = \left(\begin{array}{c||c|c} L_{00}^{(k)} & 0 & 0 \\ \hline\hline l_{10}^{(k)\,T} & \lambda_{11}^{(k)} & 0 \\ \hline L_{20}^{(k)} & l_{21}^{(k)} & L_{22}^{(k)} \end{array}\right), \quad x = \left(\begin{array}{c} x_0^{(k)} \\ \hline\hline \chi_1^{(k)} \\ \hline x_2^{(k)} \end{array}\right), \quad \text{and} \quad b = \left(\begin{array}{c} b_0^{(k)} \\ \hline\hline \tilde{\beta}_1^{(k)} \\ \hline \tilde{b}_2^{(k)} \end{array}\right)$$

where $L_{00} = L_{TL}$ and the other quadrants delimited by the double lines match the rest of our generic partition described earlier. Likewise for the x and b

vectors, where $x_T^{(k)} = x_0^{(k)}$ and $b_T^{(k)} = b_0^{(k)}$. The focus of this step is to compute $x_T^{(k+1)}$. Since $x_0^{(k)}$ is known, this is done by solving for $\chi_1^{(k)}$.

Noting that $\tilde{\beta}_1^{(k)} = \beta_1^{(k)} - l_{10}^{(k)\,T} x_0^{(k)}$, and $\tilde{b}_2^{(k)} = b_2^{(k)} - L_{20}^{(k)} x_0^{(k)}$, this can be resolved in the following way.

$$Lx = \left(\begin{array}{c} L_{00}^{(k)} x_0^{(k)} \\ \hline\hline \lambda_{11}^{(k)} \chi_1^{(k)} \\ \hline l_{21}^{(k)} \chi_1^{(k)} + L_{22}^{(k)} x_2^{(k)} \end{array} \right) = \left(\begin{array}{c} b_0^{(k)} \\ \hline\hline \tilde{\beta}_1^{(k)} \\ \hline \tilde{b}_2^{(k)} \end{array} \right)$$

yielding the equalities

$$L_{00}^{(k)} x_0^{(k)} = b_0^{(k)} \tag{2.11}$$
$$\lambda_{11}^{(k)} \chi_1^{(k)} = \tilde{\beta}_1^{(k)} \tag{2.12}$$
$$L_{21}^{(k)} \chi_1^{(k)} + L_{22}^{(k)} x_2^{(k)} = \tilde{b}_2^{(k)} \tag{2.13}$$

Since $x_0^{(k)} = x_T^{(k)}$ and

$$x_T^{(k+1)} = \left(\begin{array}{c} x_0^{(k)} \\ \hline\hline \chi_1^{(k)} \end{array} \right)$$

we now require the following computed:

$$\chi_1^{(k)} = \tilde{\beta}_1^{(k)} / \lambda_{11}^{(k)} \tag{2.14}$$
$$\tilde{b}_2^{(k)} = \tilde{b}_2^{(k)} - l_{21}^{(k)} \chi_1^{(k)} \tag{2.15}$$

Similarly, we note that b_1 does not occur in the third of our equalities (2.13). It is thus safe to again overwrite b_1 with x_1. Both algorithms have the advantage of saving memory space.

At the end of this step, we have the following

$$L = \left(\begin{array}{c||c} L_{TL}^{(k+1)\,T} & 0 \\ \hline\hline L_{BL}^{(k+1)} & L_{BR}^{(k+1)} \end{array} \right) = \left(\begin{array}{c|c||c} L_{00}^{(k)} & 0 & 0 \\ \hline l_{10}^{(k)\,T} & \lambda_{11}^{(k)} & 0 \\ \hline\hline L_{20}^{(k)} & l_{21}^{(k)} & L_{22}^{(k)} \end{array} \right),$$

$$x = \left(\begin{array}{c} x_T^{(k+1)} \\ \hline\hline x_B^{(k+1)} \end{array} \right) = \left(\begin{array}{c} x_0^{(k)} \\ \hline \chi_1^{(k)} \\ \hline\hline x_2^{(k)} \end{array} \right), \quad \text{and} \quad b = \left(\begin{array}{c} b_T^{(k+1)} \\ \hline\hline b_B^{(k+1)} \end{array} \right) = \left(\begin{array}{c} b_0^{(k)} \\ \hline \beta_1^{(k)} \\ \hline\hline b_2^{(k)} \end{array} \right)$$

2.2.3 The Variants and Cases Together

Each of the four cases of the sequential single side triangular solve is dealt with in a similar way in figure (2.3). For each of these, a condensed derivation is constructed, which also provides the algorithm. There are symmetries in this four figure diagram which are worth noting.

First, we note that the computational operations in each of the four quadrants are the same, for each of the lazy and for greedy variants. Also, while the lazy operations involve an inner product followed by a scaling, the greedy one is a scaling followed by an inner product. Although the scaling is the same for all cases, the inner product differs, using a different part of the matrix each time. This means that while going around the four quadrants clockwise, we make use of portions of the matrix in a clockwise manner for the greedy algorithm and anti-clockwise for the lazy algorithm.

Secondly, consider the cases which are the transpose of one another: $Lx = b$ with $x^T U = b^T$, and $x^T L = b^T$ with $Ux = b$. In each pair, the computation follows the same direction. Moreover, the portions of the matrix used are in the transpose position from one to the other.

Figure (2.4) illustrates both the greedy and lazy versions of the lower no-transpose algorithm, with their implementation in PLAPACK. This code closely matches the morphology of the algorithm as written on paper. Once the algorithm is derived, the code in PLAPACK itself is practically written.

Once again, we note the similarity between the lazy and greedy variants. Two lines interchange: the *gemv* and the *inv scal* calls. Also, only the parameters of the *gemv* change.

These algorithms deal with the problem on an element by element approach, advancing the computation row by row, column by column. It is true that due to the data dependencies between steps in the computation, a fine granularity must be used. This approach prevents us from extracting very much, if any, parallelism. However, a coarser granularity may be used with most of the data, using a finer granularity only when required by the actual division in step (2.14).

Furthermore, when expressing these algorithms using BLAS, we have to make use of n vector-vector calls, when in fact our data is more suited to matrix-vector operations.

2.3 Blocked Algorithm

The simple algorithm detailed in the previous section has the drawback of being short-sighted. The single element approach means that it does not take advantage of the more efficient matrix-vector BLAS calls. A blocking scheme, that is, an approach which deals with blocks of the matrix and vectors at a time, allows for better performance.

Using the appropriate matrix-vector BLAS methods, available through PLAPACK, more efficient computation can take place. The blocks are no longer 1×1 as was the case when working on a single cell basis, but now $k \times k$.

The partitioning is then

$$L = \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \qquad x = \left(\begin{array}{c} x_0 \\ \hline\hline x_1 \\ \hline x_2 \end{array}\right), \qquad \text{and} \qquad b = \left(\begin{array}{c} b_0 \\ \hline\hline b_1 \\ \hline b_2 \end{array}\right)$$

where the partitioning of the central bands no longer have one of their dimensions equal to 1, but to k, which is the blocking size. This leads to the similar changes in all variants shown in Figure (2.3). The performance gain from this blocking scheme is significant, as graph (2.5) shows.

The simple implementation has a steep upward curve due to the $O(n^2)$ nature of the problem. The blocking scheme allows an increase of k in the problem size to generate a linear increase in the number of subproblems, reflecting the almost linear nature of the blocked algorithm performance at these small problem sizes. However, as the problems get much larger, the number of blocks for

Partition $U \rightarrow \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right)$, $b \rightarrow \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right)$
where U_{BR} is 0×0 and b_B is of length 0
do until U_{TL} is 0×0
Repartition

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \upsilon_{11} & \hat{u}_{12}^T \\ \hline\hline 0 & 0 & U_{22} \end{array}\right)$$

and $\left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline\hline b_2 \end{array}\right)$
where $U_{22} = U_{BR}$ and $b_2 = b_B$
Lazy: $\beta_1 \leftarrow (\beta_1 - \hat{u}_{12}^T b_2)/\upsilon_{11}$
Greedy: $\beta_1 \leftarrow \beta_1/\upsilon_{11}$
$b_0 \leftarrow b_0 - u_{01}\beta_1$
Continue with

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c||c|c} U_{00} & u_{01} & U_{02} \\ \hline\hline 0 & \upsilon_{11} & \hat{u}_{12}^T \\ \hline 0 & 0 & U_{22} \end{array}\right)$$

and $\left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline\hline \beta_1 \\ \hline b_2 \end{array}\right)$
where $U_{00} = U_{TL}$ and $b_0 = b_T$
od

(a) $Ux = b$, Upper, No-Transpose

Partition $b \rightarrow \left(\hat{b}_T^T \,\|\, \hat{b}_B^T \right)$, $U \rightarrow \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right)$
where U_{TL} is 0×0 and $\hat{b}_T^T$ is of width 0
do until U_{BR} is 0×0
Repartition
$\left(\hat{b}_T^T \,|\, \hat{b}_B^T \right) \rightarrow \left(\hat{b}_0^T \,\|\, \beta_1 \,|\, \hat{b}_2^T \right)$ and

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} U_{00} & u_{01} & U_{02} \\ \hline\hline 0 & \upsilon_{11} & \hat{u}_{12}^T \\ \hline 0 & 0 & U_{22} \end{array}\right)$$

where $U_{00} = U_{TL}$ and $\hat{b}_0^T = \hat{b}_T^T$
Lazy: $\beta_1^T \leftarrow (\beta_1^T - \hat{b}_0^T u_{01})/\upsilon_{11}$
Greedy: $\beta_1 \leftarrow \beta_1/\upsilon_{11}$
$\hat{b}_2^T \leftarrow \hat{b}_2^T - \hat{u}_{12}^T\beta_1$
Continue with
$\left(\hat{b}_T^T \,\|\, \hat{b}_B^T \right) \leftarrow \left(\hat{b}_0^T \,|\, \beta_1 \,\|\, \hat{b}_2^T \right)$ and

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \upsilon_{11} & \hat{u}_{12}^T \\ \hline\hline 0 & 0 & U_{22} \end{array}\right)$$

where $U_{22} = U_{BR}$ and $\hat{b}_2^T = \hat{b}_B^T$
od

(b) $x^T U = b^T$, Upper Transpose

Partition $L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$, $b \rightarrow \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right)$
where L_{TL} is 0×0 and b_T is of length 0
do until L_{BR} is 0×0
Repartition

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline \hat{l}_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21}^T & L_{22} \end{array}\right)$$

and $\left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline\hline \beta_1 \\ \hline b_2 \end{array}\right)$
where $L_{00} = L_{TL}$ and $b_0 = b_T$
Lazy: $\beta_1 \leftarrow (\beta_1 - \hat{l}_{10}^T b_0)/\lambda_{11}$
Greedy: $\beta_1 \leftarrow \beta_1/\lambda_{11}$
$b_2 \leftarrow b_2 - l_{21}\beta_1$
Continue with

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline \hat{l}_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21}^T & L_{22} \end{array}\right)$$

and $\left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline\hline b_2 \end{array}\right)$
where $L_{22} = L_{BR}$ and $b_2 = b_B$
od

(c) $Lx = b$, Lower No-Transpose

Partition $b \rightarrow \left(\hat{b}_T^T \,\|\, \hat{b}_B^T \right)$, $L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$
where L_{BR} is 0×0 and $\hat{b}_B^T$ is of width 0
do until U_{TL} is 0×0
Repartition
$\left(\hat{b}_T^T \,|\, \hat{b}_B^T \right) \rightarrow \left(\hat{b}_0^T \,|\, \beta_1 \,\|\, \hat{b}_2^T \right)$ and

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline \hat{l}_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array}\right)$$

where $L_{11} = L_{BR}$ and $\hat{b}_2^T = \hat{b}_B^T$
Lazy: $\beta_1^T \leftarrow (\beta_1^T - \hat{b}_2^T \hat{l}_{21}^T)/\lambda_{11}$
Greedy: $\beta_1 \leftarrow \beta_1/\lambda_{11}$
$\hat{b}_0^T \leftarrow \hat{b}_0^T - \hat{u}_{21}^T\beta_1$
Continue with
$\left(\hat{b}_T^T \,\|\, \hat{b}_B^T \right) \leftarrow \left(\hat{b}_0^T \,\|\, \beta_1 \,|\, \hat{b}_2^T \right)$

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline \hat{l}_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21}^T & L_{22} \end{array}\right)$$

where $L_{00} = L_{TL}$ and $\hat{b}_0^T = \hat{b}_T^T$
od

(d) $x^T L = b^T$, Lower Transpose

Figure 2.3: The 4 Variants of Sequential Single Side Triangular Solve

Partition $L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$, $b \rightarrow \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right)$
where L_{TL} is 0×0 and b_T is of length 0
do until L_{BR} is 0×0
Repartition

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline \hat{l}_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21}^T & L_{22} \end{array}\right)$$

and $\left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline\hline \beta_1 \\ \hline b_2 \end{array}\right)$

where $L_{00} = L_{TL}$ and $b_0 = b_T$
Greedy: $\beta_1 \leftarrow \beta_1 / \lambda_{11}$
$b_2 \leftarrow b_2 - l_{21}\beta_1$
Continue with

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline \hat{l}_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21}^T & L_{22} \end{array}\right)$$

and $\left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline\hline b_2 \end{array}\right)$

where $L_{22} = L_{BR}$ and $b_2 = b_B$
od

(a) Algorithm for greedy $Lx = b$

```
PLA_Obj_global_width( A, &totalsize );
PLA_Obj_view_all( A, &A_Btm );
PLA_Obj_view_all( b, &b_Btm );

while( k < totalsize )
{
  PLA_Obj_vert_split_2( A_Btm, (k + 1),
                        &A_BL, &A_Dummy );
  PLA_Obj_split_4( A_BL, 1, k,
                       &A_10,    &A_11,
                     &A_Dummy, &A_21);
  PLA_Obj_horz_split_2( A_Btm, 1, &A_Dummy,
                                  &A_Btm);
  PLA_Obj_horz_split_2( b_Btm, k, &b_0,
                                  &b_1 );
  PLA_Obj_horz_split_2( b_1, 1, &b_1,
                              &b_2 );
  PLA_Local_inv_scal( A_11, b_1 );
  PLA_Gemv( PLA_NO_TRANS, MinusOne,
                     A_21, b_1, One, b_2 );
  k += 1;
}
```

(b) Code for greedy $Lx = b$

Partition $L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$, $b \rightarrow \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right)$
where L_{TL} is 0×0 and b_T is of length 0
do until L_{BR} is 0×0
Repartition

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline \hat{l}_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21}^T & L_{22} \end{array}\right)$$

and $\left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline\hline \beta_1 \\ \hline b_2 \end{array}\right)$

where $L_{00} = L_{TL}$ and $b_0 = b_T$
Lazy: $\beta_1 \leftarrow (\beta_1 - \hat{l}_{10}^T b_0)/\lambda_{11}$
Continue with

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline \hat{l}_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21}^T & L_{22} \end{array}\right)$$

and $\left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array}\right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline\hline b_2 \end{array}\right)$

where $L_{22} = L_{BR}$ and $b_2 = b_B$
od

(c) Algorithm for lazy $Lx = b$

```
PLA_Obj_global_width( A, &totalsize );
PLA_Obj_view_all( A, &A_Btm );
PLA_Obj_view_all( b, &b_Btm );

while( k < totalsize )  {
 PLA_Obj_vert_split_2(A_Btm, (k + 1),
                        &A_BL, &A_Dummy);
 PLA_Obj_split_4( A_BL, 1, k,
                        &A_10,    &A_11,
                        &A_Dummy, &A_21);
 PLA_Obj_horz_split_2( A_Btm, 1, &A_Dummy,
                                 &A_Btm);
 PLA_Obj_horz_split_2( b_Btm, k, &b_0,
                                 &b_1 );
 PLA_Obj_horz_split_2( b_1, 1, &b_1,
                             &b_2 );
 PLA_Gemv(PLA_NO_TRANS, MinusOne,
                   A_10, b_0, One, b_1 );
 PLA_Local_inv_scal( A_11, b_1 );
 k += 1;
}
```

(d) Code for lazy $Lx = b$

Figure 2.4: Algorithms and PLAPACK Code

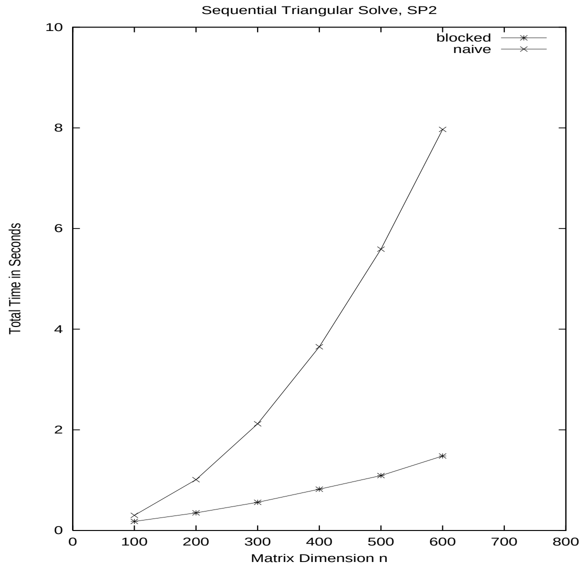

Figure 2.5: Performance of Simple and Blocked Algorithms

each particular update grows as well, again leading to a growing upward slope. However, such effects are offset by the performance gains for smaller problems.

Figure (2.6) describes the blocked algorithm along with the PLAPACK code of its lazy variant. In the code, since we no longer have individual elements with which to do the scaling, we call the previous simple implementation to solve the case of the diagonal block. This limits the computational speed up since a compromise must be reached between making the blocks larger to take advantage of the BLAS speed up and keeping blocks from growing so large that more and more time is spent in the diagonal blocks.

Partition $L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right)$, $b \rightarrow \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array} \right)$

where L_{TL} is 0×0 and b_T is of length 0

do until L_{BR} is 0×0

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array} \right)$$

and $\left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array} \right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline\hline b_1 \\ \hline b_2 \end{array} \right)$

where $L_{00} = L_{TL}$ and $b_0 = b_T$

Lazy: $b_1 \leftarrow b_1 - L_{10} b_0$

$b_1 \leftarrow L_{11}^{-1} b_1)$

Greedy: $b_1 \leftarrow L_{11}^{-1} b_1)$

$b_2 \leftarrow b_2 - L_{21} b_1$

Continue with

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array} \right)$$

and $\left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array} \right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline b_1 \\ \hline\hline b_2 \end{array} \right)$

where $L_{22} = L_{BR}$ and $b_2 = b_B$

od

(a) Algorithm Using Blocking

```
PLA_Obj_global_length( A, &totalsize );
PLA_Obj_view_all( A, &A_Btm );
PLA_Obj_view_all( b, &b_Btm );

while( k < totalsize )  {
  PLA_Obj_global_length( A_Btm, &localsize );
  if( blocksize > localsize )    {
    blocksize = localsize;
  }
  PLA_Obj_vert_split_2( A_Btm, (k + blocksize),
                     &A_BL, &A_Dummy );
  PLA_Obj_split_4( A_BL, blocksize, k,
                       &A_10,    &A_11,
                       &A_Dummy, &A_21);
  PLA_Obj_horz_split_2( A_Btm, blocksize,
                       &A_Dummy,
                       &A_Btm);
  PLA_Obj_horz_split_2( b_Btm, k, &b_0,
                                &b_1 );
  PLA_Obj_horz_split_2( b_1, blocksize, &b_1,
                                      &b_2 );
  PLA_Gemv( PLA_NO_TRANS, MinusOne,
                 A_10, b_0, One, b_1 );
  trsv1_lazy( MinusOne, One, A_11, b_1 );
  k += blocksize;
}
```

(b) Code Using Blocking (Lazy)

Figure 2.6: The Blocked Version

Chapter 3

Parallel Triangular Solve

As PLAPACK is inherently a parallel library, the PLAPACK code presented in the previous chapter can execute on a parallel computer as given. Thus, no knowledge of programming a parallel computer is needed to generate *a* parallel triangular solve. In this chapter, however, we will show that more efficient algorithms can be coded to improve performance.

3.1 Simple Parallel Implementation

There are advantages to explicitly considering how matrices are assigned to processors. Let A be a matrix partitioned as follows:

$$A = \left(\begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0(N-1)} \\ \hline A_{10} & A_{11} & \cdots & A_{1(N-1)} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{(N-1)0} & A_{(N-1)1} & \cdots & A_{(N-1)(N-1)} \end{array}\right)$$

A *Cartesian* data distribution views p processors as a $r \times c$ mesh of nodes. Rows of blocks A_{i*} are assigned to a row of processors and columns of blocks A_{*j} to columns of processors. Typically, this is done in a round-robin fashion, so that A_{ij} is assigned to processor $(i \bmod r, j \bmod c)$. PLAPACK uses a variant of this distribution, but for our discussion, the details of this are not important.

The blocking size of the algorithm and its implementation do not have to match the physical blocking size. We mentioned the diagonal block in the previous chapter: it plays an even more critical role in a parallel environment.

For our blocked algorithm in figure (2.6), an algorithmic blocking size is chosen such that this diagonal block resides on a single node. If this were not the case, then during the execution of *trsv1* , communication between multiple owner nodes must take place. This would have a number of drawbacks. First, *trsv1* a slower version of the solve and so as little time as necessary should be spent in it. There would also be more communication overhead to bring the required updated components to all the owning nodes of the diagonal block. Finally, during the computation using that diagonal block, the owning nodes have to wait for one another. Not exactly ideal for performance.

To ensure we do not have these drawbacks, the algorithmic blocking size is chosen to be at most that of the physical blocking size. For blocked algorithms like those in figure (2.6), this means that the *trsv1* call is made on data existing on a single node and executed sequentially on that node.

As the PLAPACK library is designed to make use of multicomputer machines, the code designed in the previous chapter for the straight forward blocked algorithm generates a usable program. Provided the blocking size constraints introduced are fulfilled, the code in figure (2.6) can be compiled using a parallel compiler and executed. The performance of this implementation was tested on a varying number of processors; graph (3.1) summarizes the results of these tests.

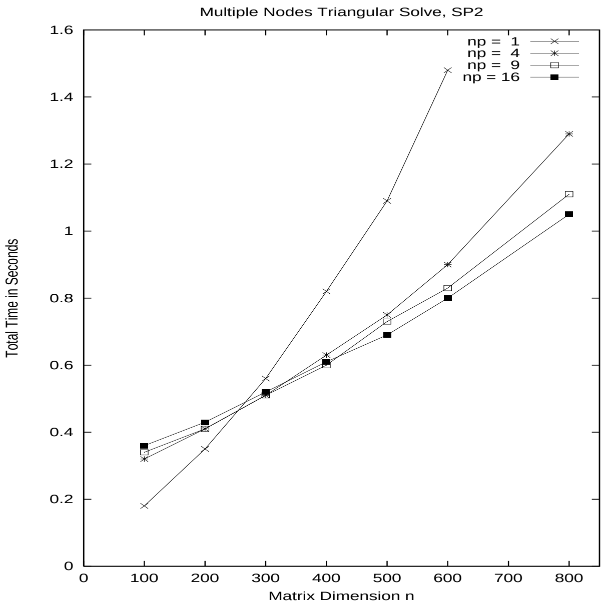

Figure 3.1: Performance Differences With Varying Number of Nodes

As we can see from this graph, porting the blocked algorithm to execute on more than one node has an immediate impact on performance. Changing the code from a sequential algorithm to a blocked algorithm also has the advantage of being flexible in terms of running adequately in both sequential and parallel environments. Since the data is separated into distinct blocks, the computation can take place in parallel when these blocks are duplicated among various processors.

On a single node, as the number of blocks which must be used in a particular step grows, so does the computation time. On multiple nodes, these blocks are duplicated. Since these blocks (excluding the diagonal triangular block) do not have data dependencies, beyond requiring a copy of the computed part of the solution vector, this computation can take place in parallel. Only the computation of the diagonal subproblem limits us and creates the data dependencies. By logically manipulating the data in blocks, a translation to the parallel world is straight forward and reaps immediate advantages.

As communication overhead is introduced, which was not present in the sequential implementation, the performance of the triangular solve on multiple processors does not do as well on smaller problems. However, this overhead is recouped in the long run when solving larger problems. With blocking size $k = 300$, all are competitive. But as k grows larger, the parallel solvers do much better. It should also be noted that as these machines have duplicated memory, increasing the number of nodes also increases the size of the problems that can be solved.

Speed-up is limited in the multiple node case also. For a fixed number of nodes, as the problem size increases, the number of blocks per node increases, increasing the computation time for a particular step. This results in the growing upward slopes in figure (3.1). And although these increasing slopes are not as steep as for the single node solver, they nonetheless get steeper.

Although adding more nodes can reduce this slope up to a point, as the graph shows, it is not a solution which increases overall performance consistently. If there are too many processors, distributing the computed portion of vector x will take an increased amount of time. Also, for a fixed problem size, as more nodes are added, more of them are idle and for longer periods of time, wasting valuable resources with little gains, if any.

The lazy and greedy variants each have a particular relationship to the problem of ensuring a balanced workload during the computation. The lazy variant, will have idle processors at the start of the computation, although by the end, it has many blocks to work with and can extract more parallelism. With the greedy variant, the reverse is true. It is during the first step that we have the most blocks to work with and at the end of the problem's computation that we have the least.

For smaller problems, we may not see much benefit to using more than one processor. Carefully choosing both a good blocking size and a suitable number of nodes is required to ensure a balanced workload and to address efficiency issues for the blocked algorithm.

3.2 Explicitly Indicating Distribution

Although PLAPACK can efficiently handle the interprocessor communication for many algorithms, it also allows the user to control it. This is done using objects called duplicated vectors, to abstract away the details linked to MPI and the particulars of the host architecture.

A duplicated vector is a number of vectors, each existing either among rows or columns of processors, globally under the name of the duplicated vector. When computation is performed using a duplicated vector, each node uses its local version and no communication takes place. Rather, it is when data is brought explicitly into or out of these duplicated vectors that communication takes place.

It is this copying into and out of the duplicated vectors, which the user contols, which specifies the communication. This enables us to fine tune the amount and timing of the communication, extracting higher performance from the algorithm.

Looking at the lazy variant, it means that we now have the following.

Repartition

$$L = \left(\begin{array}{c||c|c} L_{00}^{(k)} & 0 & 0 \\ \hline\hline L_{10}^{(k)} & L_{11}^{(k)} & 0 \\ \hline L_{20}^{(k)} & L_{21}^{(k)} & L_{22}^{(k)} \end{array}\right), \quad x = \left(\begin{array}{c} x_0^{(k)} \\ \hline\hline x_1^{(k)} \\ \hline x_2^{(k)} \end{array}\right), \quad \text{and} \quad b = \left(\begin{array}{c} b_0^{(k)} \\ \hline\hline b_1^{(k)} \\ \hline b_2^{(k)} \end{array}\right)$$

$L_{11}^{(k)}$ resides on a single node and is stored on the same column of processors as $L_{21}^{(k)}$. Also, $L_{11}^{(k)}$ is stored on the same row of processors as $L_{10}^{(k)}$. Ignoring the blocking for simplicity, an individual step in the computation is as follows:

$$\begin{aligned} \beta_1^{(k)} &\leftarrow \beta_1^{(k)} - l_{10}^{(k)\,T} x_0^{(k)} \quad (3.1) \\ \chi_1^{(k)} &\leftarrow \beta_1^{(k)} / \lambda_{11}^{(k)} \quad (3.2) \end{aligned}$$

Since our data is now duplicated, the operation $l_{10}^{(k)\,T} x_0^{(k)}$ happens on multiple nodes. However, since communication is not done explicitly, it is not optimized. This is where our duplicated vectors, *br* and *bc*, are used to improve performance.

The two duplicated vectors are partitioned in the following way:

$$br = \left(\begin{array}{c||c|c} br_0^{(k),T} & br_1^{(k),T} & br_2^{(k),T} \end{array}\right), \quad \text{and} \quad bc = \left(\begin{array}{c} bc_0^{(k)} \\ \hline\hline bc_1^{(k)} \\ \hline bc_2^{(k)} \end{array}\right)$$

A copy of vector *br* exists in every row of processors and a copy of *bc* in every column. Furthermore $bc_0^{(k)} = x_0^{(k),T}$. This in an invariant at the start of each iteration.

This implies that on a node which owns a part of $l_{10}^{(k)\,T}$ we guarantee that there is the required part of $x_0^{(k)}$ also, stored in $bc_0^{(k)}$. The computation can now be expressed in the following way.

$$\begin{aligned} br_1^{(k)} &\leftarrow l_{10}^{(k)\,T} bc_0^{(k)} \quad (3.3) \\ \beta_1^{(k)} &\leftarrow br_1^{(k)} \quad (3.4) \\ \chi_1^{(k)} &\leftarrow \beta_1^{(k)} / \lambda_{11}^{(k)} \quad (3.5) \end{aligned}$$

Operation (3.3) is explicitly done locally. No communication takes place during that step; the result is also stored locally in the other duplicated vector. Communication takes place explicitly in operation (3.4), where the local contributions stored in the duplicated vector are gathered together (summing them) and subtracted from $\beta_1^{(k)}$.

This step is not yet finished. For the next step in the computation to be able to take place, we must validate the invariant. This is done by communicating the result of this step to the other processors which will need it. That is, the nodes in the same column of the mesh.

$$\begin{aligned} br_1^{(k)} &\leftarrow l_{10}^{(k)\,T} bc_0^{(k)} \quad (3.6) \\ \beta_1^{(k)} &\leftarrow br_1^{(k)} \quad (3.7) \\ \chi_1^{(k)} &\leftarrow \beta_1^{(k)} / \lambda_{11}^{(k)} \quad (3.8) \\ bc_1^{(k)} &\leftarrow \chi_1^{(k)} \quad (3.9) \end{aligned}$$

With operation (3.9), communication is now explicitly defined in whole. We also have ensured that all communication was done only in a particular column or row of processors at any time. By describing the communication in the algorithm, we have ensured that all the computation would take place locally, without hidden communication overhead.

PLAPACK provides the functionality required to execute (3.4) and (3.9), as is shown in figure (3.2). The code demonstrates the use of two duplicated vectors. As this is a blocked algorithm, we make a call to a local solve. This is effectively a duplicate of figure (3.2), although working at the element level of that particular block. We make use of the explicit communication there also.

```
while( k < totalsize )
{
  [...]
  PLA_Obj_horz_split_2( bc, k, &bc_0, &bc_1 );
  PLA_Obj_horz_split_2( bc_1, blocksize, &bc_1,
                                         &bc_2 );

  PLA_Obj_vert_split_2( br, k, &br_0, &br_1 );
  PLA_Obj_vert_split_2( br_1, blocksize,
                                 &br_1, &br_2 );

  PLA_Local_gemv( PLA_NO_TRANS, MinusOne,
                      A_10, br_0, One, bc_1 );
  PLA_Reduce_x( PLA_SHAPE_GENERAL,
                            bc_1, One, b_1 );
  PLA_Local_trsv( PLA_LOWER_TRIANGULAR,
              PLA_NO_TRANS, diag, A_11, b_1 );
  PLA_Copy( b_1, br_1 );
  [...]
}
```

(a) Code with Explicit Communication

```
The duplicated vectors bc and
br are partitioned in the same
way as vector b.

This allows us to make gemv call
a local one easily and in
parallel. These contributions
can then be summed up together.

Then we solve the diagonal edge
as a local solve.

An update of the duplicated
vector br is then made to
maintain the loop invariants.
```

(b) Comments for the Code

Figure 3.2: Blocked Version with Explicit Communication

3.3 Outer Blocking Algorithm

Previously, we have let the physical properties of the machine and nodes dictate the blocking; we made sure that the diagonal blocks could fit on one node. Although this has yielded adequate results, increasing the blocking size, especially on larger problems, is beneficial.

Furthermore, our distribution and gathering as presented in figure (3.2) effectively sequentialize the computation between the rows and columns of nodes. Even nodes in different rows or columns which could start computation on their local contributions can be tied up waiting for the results of the latest portion of the solution vector.

The concept of an outer blocking size allows us to go beyond the past algorithms. This outer blocking size is larger than the physical blocking size of the data distribution among the nodes, ideally some multiple of it. This outer blocking is in combination to our previous blocking which was itself based on the physical distribution. Computation is then done logically among these larger blocks while allowing for the finer granularity required when dealing with the diagonal blocks.

These greater blocks reduce the number of communication start-ups required. By including more than one node at a time, we also guarrantee a more balanced workload. In fact, when we are not near the start or the end of the computation, where it is no longer possible to use all available processors, an outer blocking size can keep all nodes busy.

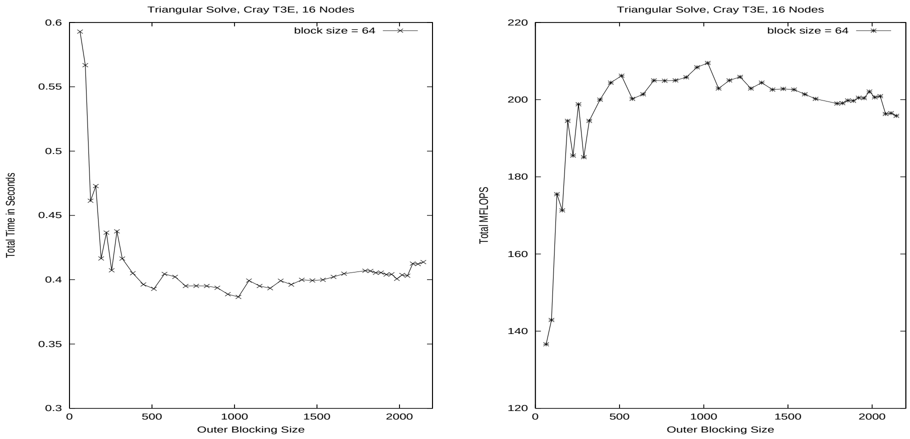

Figure 3.3: Varying Outer Blocking Size with Problem Size = 9000

Figure (3.3) shows the effect of varying the blocking size on the same problem. For these tests, the problem size was 9000×9000 and the physical blocking size was 64. The physical blocking size of 64 was used because a performance analysis showed it yielded the best performance for this problem. A number of things come to light from this graph, which are critical in obtaining high performance from the algorithms.

First, an outer blocking size that is equal to the physical blocking size (equivalent to no outer blocking) yields the poorest results. Even a slightly larger outer blocking improves performance.

Secondly, the zig-zag pattern of the graph. This indicates that certain multiples of the physical blocking size yield higher performances than neighboring ones. Looking at the graph from left to right, as we increase the outer blocking size, the highest performances are obtained in turn using 128, 256, 512 and then 1024 as the outer blocking size. The peak for this particular problem was achieved with an outer blocking size of 1024, yielding 210 MFLOPS.

Thirdly, it is interesting to note that the next local performance peak at 2048, is actually lower than the peak at 1024. In fact, as figure (3.3b) shows more clearly, the maximum performance has been reached and increasing the outer blocking size no longer improves performance. There are two extreme cases for the outer blocking size. If it is chosen to be equal to the problem size, it has no benefit. The other extreme is to choose an outer blocking size equal to the physical blocking size, which is again redundant and has no benefit. They are effectively algorithmically equivalent and in both cases, performance equally low. This was not surprising but to be expected and an intermediate size yields better performance.

3.4 Eliminating Unnecessary Communication

Although using an outer block size does reduce the number of times we must initiate communication between nodes, reducing communication overhead, this has tangible effects only at larger problem sizes. We can makes other gains. There is still some communication which can be eliminated, or merged together.

Examples of this are the *reduce* and *copy* operations. These are made both in the block-level and the element-level solve functions. We do not need the *reduce* call at the higher level, since the others will in effect make it redundant. In the same way, since we need to make the *copy* each time at the lower level, duplicating it at the higher level is also wasteful.

Figure (3.4) illustrates the coding of both the block and element level functions for this refined algorithm. The code for the partitioning of the various objects was omitted for conciseness since it repeated that of figure (3.2).

```
while( k < totalsize )
{
  [...]
  PLA_Local_gemv( PLA_NO_TRANS,
      MinusOne, A_10, br_0, One, bc_1 );
  PLA_Local_trsv1( PLA_LOWER_TRIANGULAR,
      PLA_NO_TRANS, diag, A_11, b_1 );
  k += blocksize;
}
```

(a) Blocked Code, Red. Comm.

```
while( k < totalsize )
{
  [...]
  PLA_Local_gemv( PLA_NO_TRANS,
      MinusOne, A_10, br_0, One, bc_1 );
  PLA_Reduce_x( PLA_SHAPE_GENERAL,
                     bc_1, One, b_1 );
  PLA_Local_inv_scal( A_11, b_1 );
  PLA_Copy( b_1, br_1 );
  k += 1;
}
```

(b) Element Code, Red. Comm.

Figure 3.4: The Full Blocked Version with Streamlined Communication

Figure (3.5) shows the time and MFLOPS performance of such an implementation, using an outer blocking size and reduced communications, on the Cray T3E, with an increasing number of nodes.

Although performance improves with additional nodes, there is a linear limit we do not break. Just as with other graphs, the slope of the time graph increases as the problem size increases. This is especially noticable with smaller number of nodes. However, it is worthwhile to note that the slope increases at a slower rate than previously observed.

3.5 Optimizing Communication

In each of our cases, the flow of communication in the node mesh is always down-right or up-left, throught-out the computation. Since our communication is performed in rows or columns only, we can make use of this consistency in organizing our communication in the mesh.

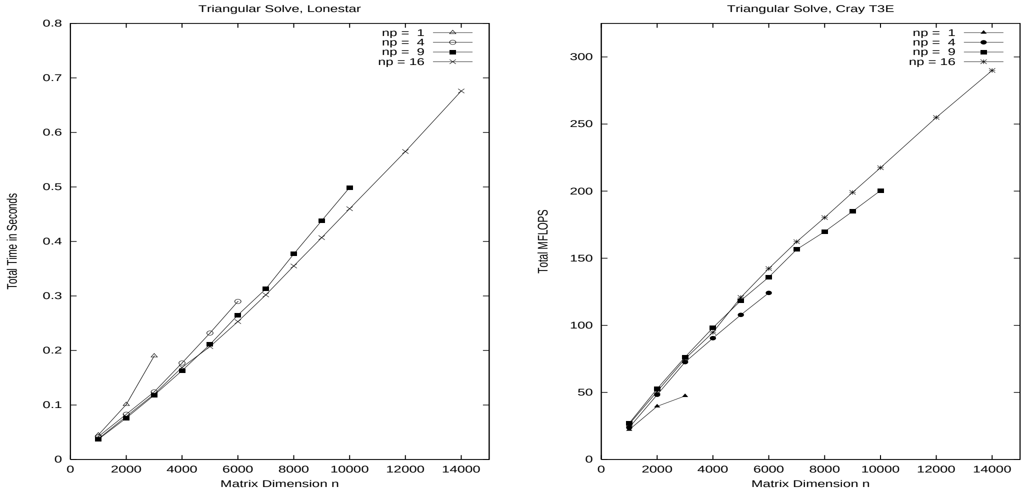

Figure 3.5: Outer Blocked Algorithm: Time and MFLOPS

By initializing the parameters of the communication to ensure it is done only in a particular direction, communication costs can be reduced further.

3.6 Performance

Having improved the performance of the algorithm in incremental steps, we have gained extra computational throughput each time. We can now look back and provide some perspective on the performance of this optimized algorithm and the logical progression of its refinement. Figure (3.6) contrasts the performance of the straight forward blocked parallel algorithm originally in the PLAPACK library, against the most efficient algorithm we obtained, making use of explicit communication, outer blocking and reduced communication.

The performance improvement is significant. On large problems, the old algorithm is over three times slower than the new one. The original PLAPACK algorithm did not make use of all these optimizing techniques.

Although for small problems, optimization provides only small performance improvements, each small optimization becomes significant when solving large problems. This is because the number of steps is increased and the penalities accumulate. The optimized algorithm also has the added advantage of being less dependent on the physical blocking size of the problem. The outer blocking and the reduced communication we implement ensure that the new version outperformes the original one under variable conditions.

It should also be noted that the incremental approach used in optimizing the algorithm was made using specific concepts and applying them to our problem. This allows us to apply these concepts to obtain better performance on other problems in the future.

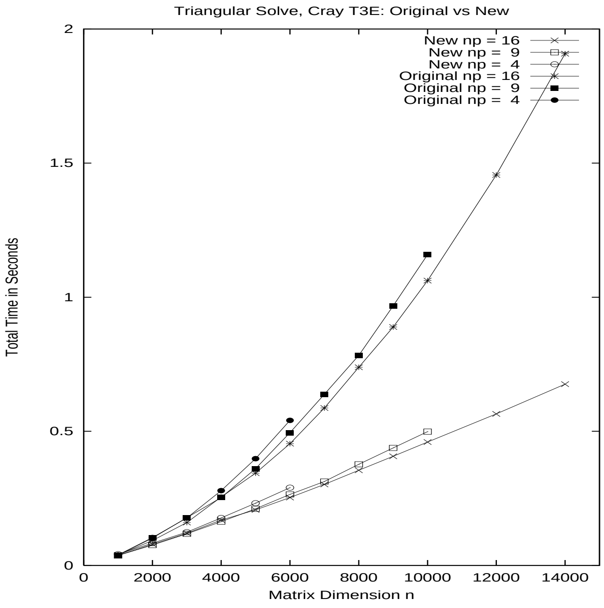

Figure 3.6: Final Performance Comparison

3.7 Going Further

The next step forward for this algorithm is to generalize it to include the cases of the multiple side triangular solve[1]. As always, there is a simple way of doing this. We can extend the single side solver to the solve multiple side problems by executing the single side solver multiple times. Although this falls far short of an efficient solution, the gains achieved in the single side solver would cary over. However, for problems with vectors of many sides, the gains are diluted.

We can apply the same refinement steps as used above to come up with a better implementation yet. The blocking strategy motivates us to solve portions of each of the multiple sides at each step. By reusing portions of the matrix, through caching and more efficient calls (BLAS2 rather than BLAS1) to solve the diagonal blocks, we can improve overall performance.

The next step is to realize that if the vectors are wider that longer, we can extract more parallelism through the redistribution of the matrix rather than the vector through the duplicated objects. Explicitly communicating the smaller objects rather than the larger ones, we again make performance gains.

[1] A draft paper was written on this subject the previous year, which showed the need to work from a clearly derived, efficient single side triangular solve

Chapter 4

Conclusion

The triangular solve is often used in conjunction with other linear algebra operations in larger problems, such as the Cholesky factorization. These are $O(n^3)$ computational problems and the triangular solve rarely accounts for more than ten percent of the overall computation time of such problems.

Through methodical optimization however, the performance of the triangular solve can be improved significantly. The three-fold performance gain of our optimized final algorithm results in saving six to seven percent of the overall computation time of the Cholesky. A visible and worthwhile improvement.

We have also shown that there is an exploitable symmetry between the greedy and the lazy variants. It is possible to construct a more complex hybrid of the two, taking advantage of their different load balancing at intermediate stages of the computation. This has the potential of yielding high performance more consistently, in situations where a purely one variant approach would see its performance degrade.

The progressive derivation we used also makes it possible to develop a multiple right hand side triangular solve from this single side solver, by following the same steps of refinement. It also enables us to tackle issues arising from the particular nature of the multiple side triangular solve by applying the concepts and optimization techniques discussed for the single side triangular solve. This paper provides more than just the foundations in working towards a high performance solution of such a problem, it also provides an effective framework.

It is the use of a clean abstraction which makes all this possible. Deriving a simple, mathematically correct algorithm which will implement a triangular solve is straight forward. With a user-friendly library like PLAPACK, it does not required great expertise to translate an algorithm into a working parallel code, capable of obtaining decent result. However, the abstraction PLAPACK provides makes it possible to go further yet.

This abstraction does not only translate into shorter coding times or simpler formal verification of the implementation. It means that more complicated and efficient algorithms, yielding increased performance, can now be implemented with relative ease. In the original BLAS, a plethora of indices had to be used but this new simplicity allows for the research and refinement of deeper concepts, providing a doorway to the development of improved algorithms.

Bibliography

[1] Michael T. Heath and Charles H. Romine, "Parallel Solution of Triangular Systems on Distributed-Memory Multiprocessors", *SIAM J. Sci. Statist. Comput.*, Vol. 9, No. 3, May 1988, p. 558.

[2] Kuck, "Parallel Processing Of Ordinary Programs", *Advances in Computers*, 15(1976), pp. 119-179.

[3] Guangye Li and Thomas F. Coleman, "A New Method For Solving Triangular Systems on Distributed Memory Message-Passing Multiprocesors", *Tech. Rept. TR 87-812*, dept Computer Science, Cornell University, Ithaco, New York, March 1987.

[4] Guangye Li and Thomas F. Coleman, "A Parallel Triangular Solver for a Distributed Memory Multiprocessor", *SIAM J. Sci. Statist. Comput.*, Vol. 9, No. 3, May 1988.

[5] C. Moler, "Matrix Computation on Distributed Memory Multiprocesors", *SIAM*, Philadelphia, 1986, pp. 181-195.

[6] S. Eisenstat, M. Heath, S. Henkel and C. Romine, "Modified Cyclic Algorithms for Solving Triangular Systems on Distributed Memory Multiprocessors", *SIAM J. Sci. Statist. Comput.*, Vol. 9, No. 3, May 1988, p. 589.

[7] Jack Dongarra, Robert van de Geijn, David Walker, "Scalability Issues Affecting the Design of a Dense Linear Algebra Library", *J. Parallel Distrib. Comput.*, 22(3), Sept. 1994.

[8] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, Iain Duff, "A Set of Level 3 Basic Linear Algebra Subprograms", *ACM Trans. Math. Soft.*, 16(1):1-17, March 1990.

[9] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, Richard J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms", *ACM Trans. Math. Soft.*, 14(1):1-17, March 1988.

[10] C.L. Lawson, R.J. Hanson, D.R. Kincaid, F.T.Krogh, "Basic Linear Algebra Subprograms for FORTRAN usage", *ACM Trans. Math. Soft.*, 5(3):308-323, Sept. 1979.

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

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

[13] L. Johnsson, "Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures", *J. Par. Dist. Comp.*, 4:133-172, 1987.