Efficient Parallel Out-of-Core Implementation of the Cholesky Factorization

Wesley C. Reiley

December 15, 1999

CS379H Honors Thesis: Efficient Parallel Out-of-Core Implementation of the Cholesky Factorization
Supervising Professor: Robert A. van de Geijn

Abstract

In this paper we describe two efficient parallel out-of-core implementations of the Cholesky factorization. We use the Parallel Out-of-Core Linear Algebra Package (POOCLAPACK) as an extension to the Parallel Linear Algebra Package (PLAPACK) to implement our out-of-core algorithms. The first algorithm uses in-core kernels with additional code to manage the I/O. This is the classical approach to out-of-core implementations of the Cholesky factorization. Our second algorithm adds an out-of-core implementation of the triangular solve with multiple right hand sides, which doesn't simply bring code in-core and run the in-core algorithm. This algorithm has the added benefit of requiring fewer copies of the matrix to be in-core at one time, thus allowing more of the matrix to be in-core at one time. Despite the extreme simplicity of POOCLAPACK and our out-of-core algorithm, the out-of-core Cholesky factorization implementation is shown to achieve in excess of 80% of peak performance on a 64 node configuration of the Cray T3E-600.

Contents

Chapter 1

Introduction

There are only a few applications left that require the solution of extremely large dense linear systems. They tend to arise from boundary-element formulations for the solution of integral equations in the areas of electro-magnetics and acoustics [5, 7, 11]. Even for those applications, much cheaper methods based on multi-pole expansions, fast multipole methods (FMM), have recently become popular [10]. Nonetheless, there are still many such applications that are solved by forming large dense systems of equations. In some cases, this is simply because the users are naive. In other cases it is a conscious decision since a considerable effort is required to reformulate the problem in a fashion that allows fast multi-pole methods to be utilized. Furthermore, there are applications requiring the solution of large linear least squares problems that also give rise to very large linear systems [2]. For applications that do still lead to large dense linear systems, the matrices involved are frequently so large that they do not fit even in the combined memories of the processors of a large distributed memory parallel supercomputer. Such problems are often referred to as out-of-core problems, since they do not fit in the core memory of the computer. The matrices are instead stored on disk.

1.1 Background

1.1.1 Distributed Memory Parallel Computing

Parallel computers consist of a collection of processors, also known as nodes. There are two major types of memory architectures 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 memory architecture is distributed memory, where each node has its 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 Cray T3E, the machine used in this thesis, follows the distributed memory model.

1.1.2 Basic Linear Algebra Subprograms

Basic Linear Algebra Subprograms[8, 9, 14], 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.

The Level 3 BLAS is very advantageous for our purposes because of its favorable operations to data ratio. For out-of-core operations, this means we will spend more time in computation then in fetching data.

1.1.3 Message Passing Interface

With the rapid advances in technology and the fast pace of change in hardware, MPI[12] was developed as a standardized communication interface for parallel computers and networks of workstations. 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.

1.1.4 PLAPACK

The Parallel Linear Algebra Package (PLAPACK) [20] is a flexible infrastructure for implementing parallel dense linear algebra routines. An MPI-like programming interface, which hides details about matrices and vectors like distribution from the user, makes both the library implementation and its use considerably simpler than more conventional packages like ScaLAPACK. In addition, the simple programming approach allows more complex algorithms to be implemented, which often yield better performance.

1.1.5 Data Distribution onto Nodes

For in-core matrices PLAPACK uses a two-dimensional Cartesian cyclic data distribution. Thus matrix B is partitioned like

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

where B_{00} is $d \times d$. The processing nodes of the parallel architecture are viewed as a logical $r \times c$ mesh of nodes, with $p = rc$. Row blocks B_{i*} and column blocks B_{*j} are all assigned to the same row and column of nodes, respectively. An over-decomposition ($N >> r, c$) is used to achieve load balance as the computation unfolds.

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 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 computation, a matrix A may be described in the following way:

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

After the computation the matrix A is described in the following way:

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

This shift in the double line essentially shows the progression of the algorithm through the object, including the direction of the progress.

1.2 Related Work

1.2.1 ScaLAPACK

The preeminent library for sequential computers and conventional (shared memory) vector supercomputers is the Linear Algebra Package (LAPACK) [1]. This package does not explicitly include out-of-core capabilities, although on machines with virtual memory the library can be used to solve problems larger than fit in-core. For larger problems, a version of this library called ScaLAPACK [4], designed for distributed memory parallel architectures, can be used. This extension of LAPACK does include prototype out-of-core implementations of some of the ScaLAPACK routines, including general linear solvers via LU factorization, positive definite linear solvers via Cholesky factorization, and linear least squares solvers via QR factorization [6]. However, this implementation does not readily allow for a full out-of-core extension.

1.2.2 SOLAR

A more serious effort to add out-of-core capabilities to LAPACK and ScaLAPACK is provided by SOLAR [18], a portable library for scalable out-of-core linear algebra computations. This library uses ScaLAPACK routines for in-core computation, but provides an I/O layer that manages matrix input-output. SOLAR achieves better I/O rates by allowing a different storage scheme for matrices on disk than is used in-core by ScaLAPACK. Impressive performance is reported for up to four nodes of an IBM SP-2. Lack of performance on larger numbers of nodes is in part blamed on non-scalability of some of the in-core parallel kernels used.

1.2.3 Other Out-of-Core Efforts

It should be noted that the above described parallel out-of-core library efforts are in addition to a number of parallel out-of-core implementations of individual operations or machine specific libraries for dense linear systems reported in the literature [2, 13, 3, 16, 17]. Additional references to applications requiring large dense linear solves are given in [5, 7, 11]. Additional references to research using fast summation methods like FMM are given in [10].

1.3 Contributions of this study

The primary contribution is the out-of-core infrastructure, POOCLAPACK, that we added to PLAPACK. This extension allows out-of-core routines to be developed very easily when combined with PLAPACK. We developed POOCLAPACK on the Cray T3E architecture, but it is easily portable[15].

We introduced an out-of-core implementation of the Cholesky factorization that varies from the classical out-of-core implementation. Our new one-tile approach allows more of the matrices to be in-core during the out-of-core operation. This improves memory utilization, and improves the number of operations to disk I/O ratio.

Our implementation of the Cholesky factorization led to the introduction of three common out-of-core linear algebra subroutines: the symmetric rank-k update, matrix-matrix multiplication, and triangular solve with multiple right hand sides. These subroutines are built in such a way that they can be readily re-used in future implementations of other linear algebra routines.

1.4 Organization of this Paper

This paper is organizes as follows: Chapter 2 introduces two algorithms for solving the Cholesky factorization. Chapter 3 discusses the out-of-core implementations of of the Cholesky factorization. This chapter also introduces the out-of-core I/O library, POOCLAPACK, which is used in our out-of-core Cholesky factorization implementations. Chapter 4 discusses the performance attained for the different implementations of the Cholesky factorization. Concluding remarks and future directions are given in Chapter 5.

1.5 Acknowledgements

It is the in-core work on PLAPACK that has made this thesis possible. I am especially grateful to Professor Robert van de Geijn for his patience and help on selected problems encountered throughout this project. I appreciate the time Professor Alan Cline has spent as my second reader. I would like to thank Professor Mohamed Gouda, as the honors advisor, for his support of this thesis.

Access to equipment for development of the described infrastructure was provided by the National Partnership for Advanced Computational Infrastructure (NPACI) and The University of Texas Advanced Computing Center (TACC). We also gratefully acknowledge access to the Cray T3E-600 System at the Goddard Space Flight Center provided by the NASA HPCC Earth and Space Science Project.

Chapter 2

Cholesky Factorization

Given an $n \times n$ symmetric positive definite matrix A, its Cholesky factorization is given by $A = LL^T$ where L is a lower triangular matrix. In this section, we develop two different algorithms for this operation, a *right-* and a *left-*looking algorithm, using LAPACK terminology. While the right-looking algorithm is more appropriate for an (parallel) in-core implementation, the left-looking algorithm has known advantages for an out-of-core implementation. We will develop blocked versions of the algorithm, since these are known to yield better ratios of the number of computations to memory operations, thus allowing better utilization of hierarchical memories.

2.1 Right Looking Variant

The right-looking algorithm for implementing this operation can be described by partitioning the matrices

$$A = \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right) \quad \text{and} \quad L = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right)$$

where A_{11} and L_{11} are $b \times b$ sub-matrices. The $\star$ indicates the symmetric part of A, which will not be updated. Now,

$$A = \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} L_{11}^T & L_{21}^T \\ \hline 0 & L_{22}^T \end{array}\right) = \left(\begin{array}{c|c} L_{11}L_{11}^T & \star \\ \hline L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{array}\right)$$

From this we derive the equations

$$\begin{array}{rcl} A_{11} & = & L_{11}L_{11}^T \\ A_{21} & = & L_{21}L_{11}^T \\ A_{22} - L_{21}L_{21}^T & = & L_{22}L_{22}^T \end{array}$$

An algorithm for computing the Cholesky factorization is now given by

1. Partition $A = \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right)$
2. $A_{11} \leftarrow L_{11} = \text{Chol.fact.}(A_{11})$
3. $A_{21} \leftarrow L_{21} = A_{21}L_{11}^{-T}$
4. $A_{22} \leftarrow A_{22} - L_{21}L_{21}^T$

5. Continue recursively with A_{22}

Note that only the upper or lower triangular part of a symmetric matrix needs to be stored and the above algorithm only updates the lower portion of the matrix with the result L. As a result, in the step $A_{22} \leftarrow A_{22} - L_{21}L_{21}^T$ only the lower portion of A_{22} is updated, which is typically referred to as a *symmetric rank-k update* (with $k = b$).

One question that may be asked about the above algorithm is what is stored in the matrix after a number of recursive steps. We answer this by partitioning

$$A = \left(\begin{array}{c||c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \tag{2.1}$$

where A_{TL} and L_{TL} are $k \times k$. Here "TL" , "BL", and "BR" stand for "Top-Left", "Bottom-Left", and "Bottom-Right", respectively. As seen before

$$A = \left(\begin{array}{c||c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \left(\begin{array}{c||c} L_{TL}^T & L_{BL}^T \\ \hline\hline 0 & L_{BR}^T \end{array}\right) = \left(\begin{array}{c||c} L_{TL}L_{TL}^T & \star \\ \hline\hline L_{BL}L_{TL}^T & L_{BL}L_{BL}^T + L_{BR}L_{BR}^T \end{array}\right) \tag{2.2}$$

so that

$$\begin{aligned} A_{TL} &= L_{TL}L_{TL}^T & (2.3)\\ A_{BL} &= L_{BL}L_{TL}^T & (2.4)\\ A_{BR} &= L_{BR}L_{BR}^T + L_{BL}L_{BL}^T & (2.5) \end{aligned}$$

It can be easily verified that the above algorithm maintains the conditions

- A_{TL} has been overwritten by L_{TL},
- A_{BL} has been overwritten by L_{BL}, and
- A_{BR} has been overwritten by $A_{BR} - L_{BL}L_{BL}^T$.

while at each step increasing the size of A_{TL} by b. Thus, the matrix with which the algorithm is continued at each step is the sub-matrix $A_{BR}$and to complete the Cholesky factorization, it suffices to compute the factorization of the updated A_{BR}. This motivates the algorithm given in Fig. 2.1.

2.2 Left Looking Variant

To derive a *left-looking* variant for computing this factorization, consider again Eqns. (2.1)–(2.5). This time assume that at the current stage

- A_{TL} has been overwritten by L_{TL},
- A_{BL} has been overwritten by L_{BL}, and
- A_{BR} *has not been changed*

To derive an algorithm that maintains this condition, while moving the computation ahead, repartition

$$A = \left(\begin{array}{c||c|c} A_{00} & \star & \star \\ \hline\hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) \quad \text{and} \quad L = \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) \tag{2.6}$$

where $A_{00} = A_{TL}$ and $L_{00} = L_{TL}$. Notice that

$$A = \left(\begin{array}{c||c|c} A_{00} & \star & \star \\ \hline\hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) \left(\begin{array}{c||c|c} {L_{00}}^T & {L_{10}}^T & {L_{20}}^T \\ \hline\hline 0 & L_{11}^T & L_{21}^T \\ \hline 0 & 0 & L_{22}^T \end{array}\right) \tag{2.7}$$

Since

$$\begin{array}{rcl} A_{11} & = & L_{10}L_{10}^T + L_{11}L_{11}^T \\ A_{21} & = & L_{20}L_{10}^T + L_{21}L_{11}^T \end{array}$$

and realizing that A_{10} has been overwritten by L_{10} and A_{20} has been overwritten by L_{20}, we find that the following computations compute L_{11} and L_{21}:

$$\begin{array}{rcl} A_{11} & \leftarrow & L_{11} = \text{Chol.fact.}(A_{11} - L_{10}L_{10}^T) \\ A_{21} & \leftarrow & L_{21} = (A_{21} - L_{20}L_{10}^T)L_{11}^{-T} \end{array}$$

The algorithm for the left-looking version of Cholesky factorization is now given in Fig. 2.2.

2.3 Sequential Implementation

Either of the two algorithms presented can be used for efficient sequential in-core implementation of the Cholesky factorization. In practice, the right-looking algorithm is favored for a rather curious reason: The bulk of the computation in the right-looking algorithm is in the rank-k update $A_{22} \leftarrow A_{22} - L_{21}L_{21}^T$ and for the left-looking algorithm in the matrix-matrix multiply $A_{21} \leftarrow A_{21} - L_{20}L_{10}^T$. While there is no technical reason for this, the level-3 BLAS [8] kernel `□syrk` that implements the symmetric rank-k update tends to achieve higher performance than the matrix-matrix multiply kernel `□gemm` *for the special case where one of the matrices is transposed*. From our experience, we believe the reason is that the symmetric rank-k update is a modification of the general rank-k update, which is at the heart of fast implementations of the LINPACK benchmark. Vendors tend to pay a lot of attention to this kernel since it is key to the performance on the benchmark. Some vendors tend to spend less time optimizing other cases of the matrix-matrix multiply, while other vendors pride themselves on delivering highly optimized versions of all BLAS. Packages like LAPACK favor the right-looking variants of these kinds of algorithms.

2.4 In-Core Parallel Implementation

The in-core parallel implementation we will be using throughout this study is the PLAPACK Cholesky factorization `PLA_Chol` routine. This routine implements the right-looking algorithm due to its favorable performance advantages. The details of the implementation is beyond the scope of this thesis [19].

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

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

$A_{11} \leftarrow L_{11} = \text{Chol.fact.}(A_{11})$
$A_{21} \leftarrow L_{21} = A_{21} L_{11}^{-T}$
$A_{22} \leftarrow A_{22} - L_{21} L_{21}^T$
continue with

$$\left(\begin{array}{c\|c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} A_{00} & \star \\ \hline A_{10} & A_{11} \\ \hline\hline A_{20} & A_{21} \end{array}\right)$$

enddo

Figure 2.1: Blocked right-looking Cholesky factorization algorithms.

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

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

$A_{11} \leftarrow A_{11} - A_{10} A_{10}^T$
$A_{21} \leftarrow A_{21} - A_{20} A_{10}^T$
$A_{11} \leftarrow L_{11} = \text{Chol.fact.}(A_{11})$
$A_{21} \leftarrow L_{21} = A_{21} L_{11}^{-T}$
continue with

$$\left(\begin{array}{c\|c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} A_{00} & \star \\ \hline A_{10} & A_{11} \\ \hline\hline A_{20} & A_{21} \end{array}\right)$$

enddo

Figure 2.2: Blocked left-looking Cholesky factorization algorithms.

Chapter 3

Out-of-Core Parallel Cholesky Factorization

Our out-of-core implementations will be based on the left-looking algorithm presented in Section 2.2. There are two basic reasons for this: First, the left-looking Cholesky requires approximately half the I/O operations of the right-looking algorithm. Second, it is easer to add *check-pointing* to a left-looking algorithm. Check-pointing allows for a restart partially into the computation in case of a system failure. Before we can discuss our out-of-core implementations, we must discuss the out-of-core library that we use for our implementations.

3.1 POOCLAPACK

The Parallel Out-of-Core Linear Algebra Package (POOCLAPACK) is an extension to PLAPACK that allows out-of-core problems to be solved in the same convenient manner in which in-core problems are solved. The handling of the disk I/O is the main functionality that needs to be added in order to allow PLAPACK to handle out-of-core problems. This is done by adding file I/O routines, and implementing them in such a way that the user can copy matrices to and form disk just as he would copy matrices in memory. With this minimal additional functionality, PLAPACK is able to handle out-of-core algorithms in the same fashion as their in-core counterparts.

3.1.1 Data Distribution onto Disk

Out-of-core matrices are distributed to nodes identically to in-core matrices (Section 1.1.5), except that the data is stored in a file.

3.1.2 Flexible File Input/Output

The Cray T3E Systems have an extended IO system, called Flexible File IO (FFIO). This system allows the user to insert layers through which data is passed. Within the layer, the user can insert various kinds of buffers and caches. Cache and/or buffer sizes and properties, like striping across multiple disks, can be controlled by command line routines. We experimented with putting a small cache between disk and memory and used default striping settings. It should be noted that changes in the configuration of the files and cache sizes did not seem to affect performance of our algorithms much. In particular, the more sophisticated algorithms that allowed larger blocks of contiguous data to be read did not seem to be affected at all.

3.1.3 Storage by Row Panels

We must briefly discuss the storage of the matrix on disk. In-core, we will assume that the matrices are stored in column-major order. Thus, elements in columns are in contiguous memory. When reading from disk, one must consider the fact that a disk access carries a large startup cost, after which contiguous data can be read at a rate determined by the limits of the hardware. Thus, reading noncontiguous data can be costly.

While columns of matrices are in contiguous memory, reading a sub-matrix of size $t \times b$, as is encountered in the out-of-core symmetric rank-k updates described in Section 3.3.1, requires either noncontiguous data to be read or a more complex storage scheme. In our implementation, we experimented with the parallel equivalent of two storage schemes: The first stores the matrix in a file much like it would be stored in memory, in column-major order. The second partitions by row blocks of t rows each, where t is equal to the tile size discussed above. These blocks of rows are then stored in separate files. As a result, we often only need to read from one of these files, and that read is a contiguous block. For this second scheme, the Cholesky factorization views the matrix as a collection of blocks of rows.

3.2 Out-of-Core Implementation

We only describe the parallel implementations of the algorithm that uses the more complex algorithm where blocks of rows are treated as separate matrices. The primary reason is that the actual code comfortably fits on one page (Fig. 3.1 and Fig. 3.2). PLAPACK and POOCLAPACK manage complexity by hiding details of size, distribution, and storage. This approach allows us to create *views* into matrices which reference sub-matrices. Each block of t rows is passed to the routine as a view of this data.

We describe two variants of this implementation. The first variant uses two in-core tiles of data, which allows an in-core triangular solve with multiple right hand sides to be called. The second variant only uses one in-core tile of data, where we can call an out-of-core triangular solve with multiple right hand sides. The second variant allows us to use larger in-core tile sizes, but involves more I/O than the first.

3.2.1 Two-Tile Implementation

The two tile variant calls an in-core triangular solve with multiple right hand sides. We briefly describe the different parts of the routine shown in Fig. 3.1.

- The matrix is passed to the POOCLAPACK OOC Cholesky factorization as an array of N views, each of which references a panel of rows, as described in Section 3.1.3 (line 1).
- The algorithm loops over the panels, partitioning the current panel into L_{10} and A_{11} (lines 6–9).
- An in-core matrix is created to hold A_{11} and that sub-matrix is read from disk (lines 10–12). Notice that this requires only a local copy from disk to the in-core matrix.
- A parallel out-of-core symmetric rank-k update, `POOCLA_Syrk`, updates $A_{11} \leftarrow A_{11} - L_{10}L_{10}^T$ where A_{11} is in-core and L_{10} resides on disk (line 13–14). We describe this routine in more detail in Section 3.3.1
- Once updated, A_{11} is factored by a call to the parallel Cholesky factorization `PLA_Chol` (lines 15–16).
- A_{11} is written to disk, and a copy is retained in memory (lines 17–18).
- The inner-most loop updates $A_{21} \leftarrow (A_{21} - L_{20}L_{10}^T)L_{11}^{-T}$. To accomplish this, we loop over the remaining row panels, partitioning each into L_{20} and A_{21} (lines 21–23).
- An in-core matrix is created to hold A_{21} and that sub-matrix is read from disk (lines 24–26).

```
1  int POOCLA_Chol_by_panels_2_tiles( int N, PLA_Obj *A_row_panels )
2  {
3    < declarations >
4    size_done = 0;                              /* number of columns finished */
5    for ( j=0; j<N; j++ ){
6      PLA_Obj_global_length( A_row_panels[ j ], &t );     /* get tile size    */
7      /* View current L_10 and A_11 submatrices */
8      PLA_Obj_vert_split_2( A_row_panels[ j ], size_done,  &L_10, &temp );
9      PLA_Obj_vert_split_2( temp,              t,          &A_11, PLA_DUMMY );
10     /* Create an in-core matrix into which to copy A_11 */
11     PLA_Matrix_create_conf_to( A_11, &A_11_in );
12     PLA_Copy( A_11, A_11_in );
13     /* Update A_11 <- A_11 - L_10 * L_10, A_11 in-core, L_10 out-of-core */
14     POOCLA_Syrk( PLA_LOWER_TRIANG, PLA_NO_TRANS, min_one, L_10, one, A_11_in );
15     /* Factor updated in-core A_11 and write out the result */
16     PLA_Chol( PLA_LOWER_TRIANG, A_11_in );
17     /* Write out A_11 */
18     PLA_Copy( A_11_in, A_11 );
19     /* Loop over A_21 */
20     for ( i=j+1; i<N; i++ ){
21       /* View current matrices L_20 and A_21 */
22       PLA_Obj_vert_split_2( A_row_panels[ i ], size_done, &L_20_1, &temp );
23       PLA_Obj_vert_split_2( temp,              t, &A_21_1, PLA_DUMMY );
24       /* Create an in-core matrix into which to copy A_21 */
25       PLA_Matrix_create_conf_to( A_21_1, &A_21_1_in );
26       PLA_Copy( A_21_1, A_21_1_in );
27       /* Update A_21 <- A_21 - L_20 * L_10^T */
28       POOCLA_Gemm( PLA_NO_TRANS, PLA_TRANS,
29                    min_one, L_20_1, L_10, one, A_21_1_in );
30       /* Update A_21 <- L_21 = A_21 * L_11^-T */
31       PLA_Trsm( PLA_SIDE_RIGHT, PLA_LOWER_TRIANG,
32                 PLA_TRANS, PLA_NONUNIT_DIAG,
33                 one, A_11_in, A_21_1_in );
34       /* Write out A_21 */
35       PLA_Copy(A_21_1_in, A_21_1);
36       size_done += t;
37     }
38     PLA_Obj_free( &A_21_1_in );
39   }
40   < clean up >
41 }
```

Figure 3.1: POOCLAPACK Out-of-Core Cholesky factorization with 2 tiles. In this version, the matrix is presented as a collection of panels of rows in an effort to improve disk performance.

- A parallel out-of-core matrix-matrix multiplication, `POOCLA_Gemm`, updates $A_{21} \leftarrow A_{21} - L_{20}L_{10}^T$. (lines 27–29). We describe this routine in more detail in Section 3.3.2
- Once updated, A_{21} is overwritten with $L_{21} = A_{21}L_{11}^{-T}$ and written to disk. Since all operands are in-core, a call to the parallel triangular solve with multiple right hand sides `PLA_Trsm` accomplishes this task (lines 30–33).
- A_{21} is written to disk, and its in-core object is freed from memory (lines 34-38).

3.2.2 One-Tile Implementation

The one tile variant calls an out-of-core triangular solve with multiple right hand sides. We briefly describe the different parts of the routine shown in Fig. 3.2.

This implementation is very similar to the two tiled implementation, except for the following noted sections.

- In the single tiled variant, we no longer need to keep a copy of A_{11} in-core, and may free its space up immediately following the call to the parallel Cholesky factorization `PLA_Chol` (lines 15–19).
- Since we no longer have A_{11} in-core, we need to call an out-of-core parallel triangular solve with multiple right hand sides `POOCLA_Trsm` instead of the in-core `PLA_Trsm` routine (lines 31–32). We describe this routine in more detail in Section 3.3.3.

These two modifications change our two tiled variant into a one tiled variant of out implementation of the Cholesky factorization.

3.3 Out-of-Core Routines

During our discussion of the out-of-core Cholesky factorization, we found the need for a few out-of-core routines that need to be implemented. These routines are the out-of-core symmetric rank-k update, matrix-matrix multiplication, and triangular solve with multiple right hand sides. We now give a detailed analysis of these three operations.

3.3.1 POOCLA_Syrk: Symmetric Rank-K Update

We now describe in detail the out-of-core implementation of the symmetric rank-k update $A_{11} \leftarrow A_{11} - L_{10}L_{10}^T$, or, more generically, $C \leftarrow C - AA^T$.

The algorithm for implementing this operation can be described by partitioning the matrix:

$$A = \left(\begin{array}{c|c} A_L & A_R \end{array} \right)$$

Now,

$$\begin{aligned} C &= C - \left(\begin{array}{c|c} A_L & A_R \end{array} \right) \left(\begin{array}{c} A_L^T \\ \hline A_R^T \end{array} \right) \\ C &= C - A_L A_L^T - A_R A_R^T \end{aligned}$$

From this, if we assume we have reached the state where $C \leftarrow C - A_L A_L^T$ has been updated, then an algorithm for computing the symmetric rank-k update is now given by:

1. Partition $\left(\begin{array}{c|c} A_L & A_R \end{array} \right) = \left(\begin{array}{c|c|c} A_0 & A_1 & A_2 \end{array} \right)$

```
1  int POOCLA_Chol_by__panels_1_tile( PLA_Obj *A_row_panels )
2  {
3    < declarations >
4    size_done = 0;                              /* number of columns finished */
5    for ( j=0; j<N; j++ ){
6      PLA_Obj_global_length(A_row_panels[ j ], &t);  /* get tile size */
7      /* View current L_10 and A_11 submatrices */
8      PLA_Obj_vert_split_2( A_row_panels[ j ], size_done, &L_10, &temp );
9      PLA_Obj_vert_split_2( temp, t, &A_11, PLA_DUMMY);
10     /* Create an in-core matrix into which to copy A_11 */
11     PLA_Matrix_create_conf_to(A_11, &A_11_in);
12     PLA_Copy(A_11, A_11_in);
13     /* Update A_11 <- A_11 - L_10 * L_10, A_11 in-core, L_10 out-of-core */
14     POOCLA_Syrk( PLA_LOWER_TRIANG, PLA_NO_TRANS, min_one, L_10, one, A_11_in);
15     /* Factor updated in-core A_11 and write out the result */
16     PLA_Chol(PLA_LOWER_TRIANG, A_11_in);
17     /* Write out A_11 */
18     PLA_Copy(A_11_in, A_11);
19     PLA_Obj_free( &A_11_in );
20     /* Loop over A_21 */
21     for ( i=j+1; i<N; i++ ){
22       /* View current matrices L_20 and A_21 */
23       PLA_Obj_vert_split_2( A_row_panels[ i ], size_done, &L_20_1, &temp );
24       PLA_Obj_vert_split_2( temp, t, &A_21_1, PLA_DUMMY );
25       /* Create an in-core matrix into which to copy A_21 */
26       PLA_Matrix_create_conf_to(A_21_1, &A_21_1_in);
27       PLA_Copy(A_21_1, A_21_1_in);
28       /* Update A_21 <- A_21 - L_20 * L_10^T */
29       POOCLA_Gemm( PLA_NO_TRANS, PLA_TRANS, min_one, L_20_1,
30                    L_10, one, A_21_1_in);
31       /* Update A_21 <- L_21 = A_21 * L_11^-T */
32       POOCLA_Trsm( PLA_NONUNIT_DIAG, one, A_11, A_21_1_in);
33       /* Write out A_21 */
34       PLA_Copy(A_21_1_in, A_21_1);
35       size_done += t;
36     }
37     PLA_Obj_free( &A_21_1_in );
38   }
39   < cleanup >
40 }
```

Figure 3.2: POOCLAPACK Out-of-Core Cholesky factorization with 1 tile. In this version, the matrix is presented as a collection of panels of rows in an effort to improve disk performance.

```
1  int POOCLA_Syrk( int uplo, int transa,
2                   PLA_Obj alpha, PLA_Obj A_ooc,
3                   PLA_Obj beta,  PLA_Obj C )
4  {
5    < declarations >
6    < get size b, the number of columns to be read at a time >
7    /* Scale C <- beta * C */
8    PLA_Local_scal( beta, C );
9    /* A_ooc_cur view the part of A_ooc yet to be used */
10   PLA_Obj_view_all( A_ooc, &A_ooc_cur );
11
12   while ( TRUE ){
13     /* Check if part of A_ooc yet to be used is of width 0 */
14     PLA_Obj_global_width(  A_ooc_cur, &size );
15     if ( ( size = min( size, b ) ) == 0 ) break;
16     /* view current A */
17     PLA_Obj_vert_split_2( A_ooc_cur, size, &A_ooc_1, &A_ooc_cur );
18     /* Create an in-core matrix into which to copy A */
19     PLA_Matrix_create_conf_to( A_ooc_1, &A_in_1 );
20     PLA_Copy( A_ooc_1, A_in_1 );
21     /* Perform in-core symmetric rank-k update */
22     PLA_Syrk( uplo, transa, alpha, A_in_1, one, C );
23   }
24   < cleanup >
25 }
```

Figure 3.3: POOCLAPACK symmetric rank-k update routine. Matrix A, passed in as object A_ooc, is assumed to be stored on disk, and matrix C, passed in as object C, is assumed to be in-core. This version does not attempt to overlap I/O with computation.

2. $C \leftarrow C - A_1 A_1^T$
3. Continue with $\left(\begin{array}{c|c|c} A_L & \| & A_R \end{array} \right) = \left(\begin{array}{c|c|c} A_0 & A_1 & \| \; A_2 \end{array} \right)$.

A parallel implementation of this operation using POOCLAPACK is given in Fig. 3.3. We briefly describe the different parts of this routine.

- Matrices A and C are passed in as views `A_ooc` and `C`, where `A_ooc` references a matrix stored on disk, and `C` references a matrix stored in-core (line 1).
- The algorithm starts by scaling $C \leftarrow \beta C$ (line 8).
- Next, the algorithm loops over blocks of columns, partitioning off the current block A_1 as `A_ooc_1` (lines 13–17).
- An in-core matrix is created to hold A_1 and that sub-matrix is read from disk (lines 18–20).
- An in-core parallel symmetric rank-k update, `PLA_Syrk`, updates $C \leftarrow C - A_1 A_1^T$ where A_1 is in-core, referenced by `A_in_1` (line 22).

3.3.2 POOCLA_Gemm: Generic Matrix-Matrix Multiply

We now describe in detail the out-of-core implementation of the matrix-matrix multiply $A_{21} \leftarrow A_{21} - L_{20} L_{10}^T$, or, more generically, $C \leftarrow C - AB^T$.

The algorithm for implementing this operation can be described by partitioning the matrices:

$$A = \left(\begin{array}{c|c|c} A_L & \| & A_R \end{array} \right) \quad \text{and} \quad B = \left(\begin{array}{c|c|c} B_L & \| & B_R \end{array} \right)$$

Now,

$$\begin{aligned} C &= C - \left(\begin{array}{c|c|c} A_L & \| & A_R \end{array} \right) \left(\begin{array}{c} B_L^T \\ \hline\hline B_R^T \end{array} \right) \\ C &= C - A_L B_L^T - A_R B_R^T \end{aligned}$$

From this, if we assume we have reached the state where $C \leftarrow C - A_L B_L^T$ has been updated, then an algorithm for computing the matrix-matrix multiply is now given by

1. Partition $\left(\begin{array}{c|c|c} A_L & \| & A_R \end{array} \right) = \left(\begin{array}{c|c|c} A_0 & \| & A_1 \; | \; A_2 \end{array} \right)$
2. Partition $\left(\begin{array}{c|c|c} B_L & \| & B_R \end{array} \right) = \left(\begin{array}{c|c|c} B_0 & \| & B_1 \; | \; B_2 \end{array} \right)$
3. $C \leftarrow C - A_1 B_1^T$
4. Continue with $\left(\begin{array}{c|c|c} A_L & \| & A_R \end{array} \right) = \left(\begin{array}{c|c|c} A_0 & A_1 & \| \; A_2 \end{array} \right)$ and $\left(\begin{array}{c|c|c} B_L & \| & B_R \end{array} \right) = \left(\begin{array}{c|c|c} B_0 & B_1 & \| \; B_2 \end{array} \right)$

A parallel implementation of this operation using POOCLAPACK is given in Fig. 3.4. We briefly describe the different parts of this routine.

- Matrices A, B and C are passed in as views `A_ooc`, `B_ooc` and `C`, where `A_ooc` and `B_ooc` reference matrices stored on disk, and `C` references a matrix stored in-core (line 1).
- The algorithm starts by scaling $C \leftarrow \beta C$ (line 9).

```
1  int POOCLA_Gemm( int transa, int transb,
2                   PLA_Obj alpha, PLA_Obj A_ooc,
3                                  PLA_Obj B_ooc,
4                   PLA_Obj beta,  PLA_Obj C )
5  {
6    < declarations >
7    < get size b, the number of columns to be read at a time >
8    /* Scale C <- beta * C */
9    PLA_Local_scal( beta, C );
10   /* A_ooc_cur/B_ooc_cur view the part of A_ooc/B_ooc yet to be used */
11   PLA_Obj_view_all( A_ooc, &A_ooc_cur );
12   PLA_Obj_view_all( B_ooc, &B_ooc_cur );
13
14   while ( TRUE ){
15     /* Check if part of A_ooc yet to be used is of width 0 */
16     PLA_Obj_global_width(  A_ooc_cur, &size );
17     if ( ( size = min( size, nb_ooc ) ) == 0 ) break;
18     /* view current A and B */
19     PLA_Obj_vert_split_2( A_ooc_cur, size, &A_ooc_1, &A_ooc_cur );
20     PLA_Obj_vert_split_2( B_ooc_cur, size, &B_ooc_1, &B_ooc_cur );
21     /* Create an in-core matrix into which copy A and B */
22     PLA_Matrix_create_conf_to( A_ooc_1, &A_in_1 );
23     PLA_Copy( A_ooc_1, A_in_1 );
24     PLA_Matrix_create_conf_to( B_ooc_1, &B_in_1 );
25     PLA_Copy( B_ooc_1, B_in_1 );
26     /* Perform in-core matrix-matrix multiply */
27     PLA_Gemm(transa,transb, alpha, A_in_1, B_in_1, beta, C);
28   }
29   < cleanup >
30 }
```

Figure 3.4: POOCLAPACK matrix-matrix multiply routine. Matrix A and B, passed in as object A_ooc and B_ooc, are assumed to be stored on disk, and matrix C, passed in as object C, is assumed to be in-core. This version does not attempt to overlap I/O with computation.

- Next, the algorithm loops over blocks of columns, partitioning off the current block A_1 as `A_ooc_1` and B_1 as `B_ooc_1` (lines 15–20).
- In-core matrices are created to hold A_1 and B_1 and those sub-matrices are read from disk (lines 21–25).
- A in-core parallel matrix-matrix multiply, `PLA_Gemm`, updates $C \leftarrow C - A_1 B_1^T$ where A_1 and B_1 are in-core, referenced by `A_in_1` and `B_in_1` (line 27).

3.3.3 POOCLA_Trsm: Triangular Solve with Multiple Right Hand Sides

We now describe in detail the out-of-core implementation of the triangular solve with multiple right hand sides $L_{21}L_{11}^T = A_{21}$, or, more generically, $CL^T = B$, overwriting B with C.

The algorithm for implementing this operation can be describe by partitioning the matrices:

$$C = \left(\begin{array}{c||c} C_L & C_R \end{array}\right) \quad \text{and} \quad L = \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \quad \text{and} \quad B = \left(\begin{array}{c||c} B_L & B_R \end{array}\right)$$

Now,

$$\begin{array}{rcl} \left(\begin{array}{c||c} C_L & C_R \end{array}\right) \left(\begin{array}{c||c} L_{TL}^T & L_{BL}^T \\ \hline\hline 0 & L_{BR}^T \end{array}\right) & = & \left(\begin{array}{c||c} B_L & B_R \end{array}\right) \\ \left(\begin{array}{c||c} C_L L_{TL}^T & C_L L_{BL}^T + C_R L_{BR}^T \end{array}\right) & = & \left(\begin{array}{c||c} B_L & B_R \end{array}\right) \end{array}$$

From this we derive the equations

$$\begin{array}{rcl} C_L L_{TL}^T & = & B_L \\ C_L L_{BL}^T + C_R L_{BR}^T & = & B_R \end{array}$$

or

$$\begin{array}{rcl} C_L & = & B_L L_{TL}^{-T} \\ C_R L_{BR}^T & = & B_R - C_L L_{BL}^T \end{array}$$

If we assume that we have reached the state where $C_L = B_L L_{TL}^{-T}$ and $B_R = B_R - C_L L_{BL}^T$ have been updated, then an algorithm for computing the triangular solve with multiple right hand sides is now given by

1. Partition $\left(\begin{array}{c||c} B_L & B_R \end{array}\right) = \left(\begin{array}{c||c|c} B_0 & B_1 & B_2 \end{array}\right)$
2. Partition $\left(\begin{array}{c||c} C_L & C_R \end{array}\right) = \left(\begin{array}{c||c|c} C_0 & C_1 & C_2 \end{array}\right)$
3. Partition $\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$
4. $C_1 \leftarrow B_1 - C_0 L_{11}^T$
5. $C_2 \leftarrow B_2 - C_1 L_{21}^T$

```
1  int POOCLA_Trsm(int diag, PLA_Obj alpha,
2                  PLA_Obj A_ooc, PLA_Obj B )
3  {
4    < declarations >
5    < get size b, the number of columns to be read at a time >
6    /* Scale B <- beta * B */
7    PLA_Local_scal( alpha, B );
8    /* A_ooc_cur/B_curr view the part of A_ooc/B to be used */
9    PLA_Obj_view_all( A_ooc, &A_ooc_cur );
10   PLA_Obj_view_all( B, &B_curr );
11
12   k = 0;
13   while ( TRUE ){
14     /* Check if part of B yet to be used is of width 0 */
15     PLA_Obj_global_width( B_curr, &size );
16     if ( 0 == ( size = min( b, size ) ) ) break;
17     /* Copy ( 0 | A_11 | A_21 )^T into the in-core A_in_1 */
18     PLA_Obj_vert_split_2( A_ooc_cur, size, &A_ooc_1, &A_ooc_cur );
19     PLA_Matrix_create_conf_to( A_ooc_1, &A_in_1 );
20     PLA_Copy( A_ooc_1, A_in_1 );
21     /* view current L_11 and L_21 */
22     PLA_Obj_horz_split_2( A_in_1, k, PLA_DUMMY, &A_in_1 );
23     PLA_Obj_horz_split_2( A_in_1, size, &L_in_11, &L_in_21 );
24     /* view current B_1 */
25     PLA_Obj_vert_split_2( B_curr, size, &B_1, &B_curr );
26     /* Solve B_1 <- C_1 = C_1 - B_1 L_11^T */
27     PLA_Trsm( PLA_SIDE_RIGHT, PLA_LOWER_TRIANGULAR,
28               PLA_TRANS, diag, one, L_in_11, B_1 );
29     /* Solve B_1 <- C_2 = C_2 - B_1 L_21^T */
30     PLA_Gemm( PLA_NO_TRANS, PLA_TRANS,
31               minus_one, B_1, L_in_21, one, B_R );
32     k += size;
33   }
34   < cleanup >
35 }
```

Figure 3.5: POOCLAPACK triangular solve with multiple right hand sides routine. Matrix A passed in as object A_ooc, are assumed to be stored on disk, and matrix B, passed in as object B, is assumed to be in-core. This version does not attempt to overlap I/O with computation.

6. Continue with $\left(\begin{array}{c|c} B_L & B_R \end{array}\right) = \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$ and $\left(\begin{array}{c|c} C_L & C_R \end{array}\right) = \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right)$ and $\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array}\right)$

A parallel implementation of this operation using POOCLAPACK is given in Fig. 3.5. We briefly describe the different parts of this routine.

- Matrices A and B are passed in as views `A_ooc` and `B`, where `A_ooc` references a matrix stored on disk, abd `B` references a matrix stored in-core (line 1).
- The algorithm starts by scaling $B \leftarrow \beta B$ (line 7).
- Next, the algorithm loops over blocks of columns of B, partitioning off the current block $\left(\begin{array}{c|c|c} 0 & L_{11} & L_{21} \end{array}\right)^T$ as `A_ooc_1` (lines 14–18).
- An in-core matrix is created to hold A_1 and that sub-matrix is read from disk (lines 19–20).
- Partition `A_ooc_1` into L_{11} and L_{21}, eliminating the extra zeros read into memory (lines 21–23).
- Partition off the current block B_1 (lines 24–25).
- An in-core parallel triangular solve with multiple right hand sides, `PLA_Trsm`, updates $C_1 = C_1 - B_1 L_{11}^T$, where B_1 is overwritten by C_1 (lines 26–28).
- An in-core parallel matrix-matrix multiply, `PLA_Gemm`, updates $C_2 = C_2 - B_1 L_{21}^T$, where B_2 is overwritten by C_2 (29–31).

Chapter 4

Performance

4.1 Testing Environment

We demonstrate performance on the Cray T3E-600 (300 MHz), with all computations performed in 64-bit arithmetic. The algorithms were implemented using an alpha release of PLAPACK Version R2.0, which performs all communication by means of MPI. We report performance measuring MFLOPS/processor (millions of floating point operations per second per processor). For reference, the following table shows performance of matrix-matrix multiplication on a single processor of the T3E-600 in MFLOPS:

n	MFLOPS
500	418
1000	443
1500	425

All performance reported in this section for the T3E-600 was measured with data streams turned on (a hardware feature that adds about 15–20% to the performance of the local matrix-matrix multiply kernel).

The Cray T3E-600 at the Goddard Space Flight Center used for the experiments has a 54 Gigabyte partition striped across 14 disks[1]. This partition was used for all I/O experiments.

4.2 Implementations tested

We report performance for six different versions of the code:

`PLA_Chol`: This version is the in-core PLAPACK Cholesky factorization.

`POOCLA_Chol`: This version views the matrix as one matrix, with each processor accessing a single file in which the local matrix is stored. The matrix is stored in this file much like an in-core matrix would be stored, i.e., it is viewed as a two-dimensional array. No effort is made to overlap I/O with computation.

`POOCLA_Chol_async`: This version is identical to `POOCLA_Chol` except that it overlaps I/O and computation during the updates $A_{11} \leftarrow A_{11} - L_{10}L_{10}^T$ and $A_{21} \leftarrow A_{21} - L_{10}L_{10}^T$.

`POOCLA_Chol_by_panels_2_tiles`: This version is given in Fig. 3.1 and views the matrix as a collection of row panels, as described in Section 3.1.3. This version uses two in-core tiles, and calls the in-core triangular solve with multiple right hand sides. No effort is made to overlap I/O with computation.

[1] the `/tmp` directory.

`POOCLA_Chol_by_panels_async`: This version is identical to `POOCLA_Chol_by_panels` except that it overlaps I/O and computation during the updates $A_{11} \leftarrow A_{11} - L_{10}L_{10}^T$ and $A_{21} \leftarrow A_{21} - L_{10}L_{10}^T$.

`POOCLA_Chol_by_panels_1_tile`: This version is given in Fig. 3.2 and views the matrix as a collection of row panels, as described in Section 3.1.3. This version uses one in-core tile, and calls the out-of-core triangular solve with multiple right hand sides. No effort is made to overlap I/O with computation.

4.3 Results

We collected two sets of results, from the two-tile implementation and from the one-tile implementation. The classic two-tile approach uses an in-core triangular solve with multiple right hand sides. This requires two in-core tiles to be present in memory at the same time, and restricts the size of the in-core tiles. The new one-tile approach uses an out-of-core triangular solve with multiple right hand sides. This only requires one in-core tile to be present in memory, and allows the size of the in-core tile to be increased. We first present our results from the many versions of the two-tile approach. We then present our results for the row panel versions of the two-tile and one-tile approaches.

4.3.1 Classical Two-Tile Approach

		tile	1×1 tiles ($n = t$)			2×2 tiles ($n = 2t$)			3×3 tiles ($n = 3t$)		
		size	MFLOPS	Time (sec)		MFLOPS	Time (sec)		MFLOPS	Time (sec)	
Algorithm	p	t	/proc.	Total	I/O	/proc.	Total	I/O	/proc.	Total	I/O
`In-core Chol`	1	2088	263	11.5							
`Chol`	1	2088	243	12.5	1.1	253	96	23	260	315	84
`Chol_async`	1	2088	257	11.8	0.4	252	96	20	266	308	61
`Chol_by_panel`	1	2088	245	12.4	1.0	296	82	9	334	245	17
`Chol_by_panel_async`	1	2088	**227**	13.3	2.0	**291**	83	8	**327**	250	12
`In-core Chol`	4	4704	304	28.5							
`Chol`	4	4704	278	31.1	2.6	183	380	182	183	1282	598
`Chol_async`	4	4704	278	31.2	2.7	176	393	182	189	1239	501
`Chol_by_panel`	4	4704	276	31.5	2.6	331	209	10	353	663	24
`Chol_by_panel_async`	4	4704	**278**	31.2	2.6	**336**	206	8	**361**	649	16
`In-core Chol`	16	8448	304	41.3							
`Chol`	16	8448	277	45.3	4.1	294	342	47	299	1135	???
`Chol_async`	16	8448	277	45.3	4.1	*	*	*	*	*	*
`Chol_by_panel`	16	8448	273	46.1	4.3	321	313	13	343	989	32
`Chol_by_panel_async`	16	8448	**277**	45.3	4.1	**326**	308	10	**347**	977	21
`In-core Chol`	64	18432	263	124							
`Chol_by_panel`	64	18432	267	122	15.0	315	827	53	331	2654	125
`Chol_by_panel_async`	64	18432	**271**	121	15.2	**317**	822	53	**339**	2594	105

Table 4.1: Performance of the various Cholesky factorization routines using In-Core TRSM on the Cray T3E-600.

First we test all of our two-tile implementations in order to determine which of the implementations performs the best. These results are reported in Table 4.1. For a fixed number of processors, we report performance for a problem

equal to the tile size $t \times t$, $(2t) \times (2t)$, and $(3t) \times (3t)$. For those familiar with PLAPACK, a distribution block size of 24 and algorithmic block size of 128 was used.

It is interesting to compare the performance of the in-core Cholesky factorization with that of the out-of-core factorizations for a $t \times t$ problem size. The moderate drop in performance illustrates the fact that $O(t^3)$ operations are being performed on $O(t^2)$ data and thus the I/O has only minor impact on performance. Recall that when the problem size n is much greater than t, this reading and writing of the tiles is amortized over even more computation. We used this observation to justify not overlapping the reading and writing of the diagonal blocks A_{11} and tiles $A_{21}^{(i)}$. As the problem size increases, the out-of-core versions yield better performance than the in-core Cholesky factorization. While on the surface this may be puzzling, notice that the larger the problem, the more computation is being performed in matrix-matrix multiplication (to update A_{21}), which executes at a higher rate of computation than the Cholesky factorization of the diagonal blocks A_{11}.

There is a noticeable improvement in performance when the specialized storage described in Section 3.1.3 is used. As predicted, the fact that the "panel" based versions read contiguous data greatly improves I/O performance. The benefits of asynchronous I/O (overlapping some of the computation with reading of data) is less dramatic. This is due to the fact that only a small percentage of execution time is being spent in I/O.

A final note: The performance numbers presented were collected on the NASA Goddard Space Flight Center Cray T3E, a heavily loaded machine where many of the applications being executed are I/O intensive. Since we did not have exclusive use of this machine, the performance reported paints a pessimistic picture. We have observed performance as high as 351 MFLOPS per processor on 64 processors for the $(3t) \times (3t)$ problem.

4.3.2 New One-Tile Approach

		tile	1×1 tiles ($n = t_l$)			2×2 tiles ($n = 2t_l$)			3×3 tiles ($n = 3t_l$)		
		size	MFLOPS	Time (sec)		MFLOPS	Time (sec)		MFLOPS	Time (sec)	
Algorithm	p	t	/proc.	Total	I/O	/proc.	Total	I/O	/proc.	Total	I/O
`Chol_by_panel 2-tile`	1	t_s=2088	268	39.4	4.1	311	272	21	347	823	53
`Chol_by_panel 1-tile`	1	t_l=3168	**267**	39.8	5.4	**331**	256	19	**354**	809	47
`Chol_by_panel 2-tile`	4	t_s=4704	297	80	5.2	350	546	21	362	1782	71
`Chol_by_panel 1-tile`	4	t_l=6600	**311**	77.1	5.9	**361**	532	25	**365**	1771	122
`Chol_by_panel 2-tile`	16	t_s=8448	289	124	7.8	339	850	29	337	2882	163
`Chol_by_panel 1-tile`	16	t_l=12000	**315**	114	9.0	**355**	812	39	**364**	2674	134
`Chol_by_panel 2-tile`	64	t_s=18432	272	342	32	294	2545	348			
`Chol_by_panel 1-tile`	64	t_l=26160	**306**	305	33	**323**	2316	299			

Table 4.2: Performance of the various Cholesky factorization routines using Out-of-Core TRSM on the Cray T3E-600. The test sizes of these matrices were based on the 1-tile tile size.

Once we have determined the best two-tile implementation, we compare the two-tile and one-tile versions of that implementation, which was the row panel version. We assume that the one-tile version of the implementation will also perform the best of the one-tile implementations. We report two sets of performance results. The first set of results determines the problem size from the one-tile tile size. These results are reported in Table 4.2. The second set of results determines the problem size from the two-tile tile size. These results are reported in Table 4.3. For a fixed number of processors, we report performance for a problem equal to the respective tile size $t \times t$, $(2t) \times (2t)$, and $(3t) \times (3t)$. We use the same distribution and algorithmic block sizes.

It is interesting to look at the two tables, Table 4.2 and Table 4.3, and notice which version performs better. The results where the one-tile tile size is used, Table 4.2, show that the one-tile version performs better than the two-tile

Algorithm	p	tile size t	1×1 tiles ($n = t_s$) MFLOPS /proc.	Time (sec) Total	I/O	2×2 tiles ($n = 2t_s$) MFLOPS /proc.	Time (sec) Total	I/O	3×3 tiles ($n = 3t_s$) MFLOPS /proc.	Time (sec) Total	I/O
`Chol_by_panel 2-tile`	1	t_s=2088	245	12.4	1.0	296	82	9	334	245	17
`Chol_by_panel 1-tile`	1	t_l=3168	**222**	13.7	2.1	**275**	88	11	**329**	248	19
`Chol_by_panel 2-tile`	4	t_s=4704	276	31.5	2.6	331	209	10	353	663	24
`Chol_by_panel 1-tile`	4	t_l=6600	**261**	33.1	3.4	**319**	218	17	**345**	680	37
`Chol_by_panel 2-tile`	16	t_s=8448	273	46.1	4.3	321	313	13	343	989	32
`Chol_by_panel 1-tile`	16	t_l=12000	**275**	45	4.3	**320**	313	18	**331**	1026	64
`Chol_by_panel 2-tile`	64	t_s=18432	267	122	15.0	315	827	53	331	2654	125
`Chol_by_panel 1-tile`	64	t_l=26160	**270**	121	16	**296**	881	81	**253**	3482	759

Table 4.3: Performance of the various Cholesky factorization routines using Out-of-Core TRSM on the Cray T3E-600. The test sizes of these matrices were based on the 2-tile tile size.

version. However, The results where the two-tile tile size is used, Table 4.3, show that the two-tile version performs better than the one-tile version. This is most easily explained by the fact that each version performs better when its problem size is a multiple of its respective tile size. This allows each in-core iteration through the problem to use the maximum amount of available memory. However, when one version is running with a problem size that is not a multiple of its tile size, the last iteration will not fill up memory, and hence will not achieve its peak performance. This is especially the case since at max three iterations are computed. For these results, when one version is run with a problem size that is not a multiple of its tile size, at max two iterations are running at peak, while the third is running below peak performance. In order to overcome this situation, much larger problem sizes would need to be tested. However, due to lack of disk space on our test machines, we were limited to the sizes of our problems sets.

Chapter 5

Conclusions

The applications which use the Cholesky factorization routinely have large data sets. These large data sets often do not fit into the in-core memory of a machine, and hence an out-of-core solution must be used. An efficient parallel out-of-core Cholesky factorization would allow these data sets to be processed in a timely manner regardless of memory limitations. Such an implementation would not only process the data faster, but gives the ease of a library call.

The Parallel Out-of-Core Linear Algebra Package (POOCLAPACK) allowed us to implement an out-of-core Cholesky factorization in a timely and efficient manner. The constructs provided by POOCLAPACK allow us to refine our implementation to create a highly efficient and simple approach to I/O.

We developed out-of-core implementations of three routines, that are used in our out-of-core Cholesky factorization implementations: symmetric rank-k update, matrix-matrix multiplication, and triangular solve with multiple right hand sides. These three out-of-core implementations allow us to create two out-of-core Cholesky factorization implementations.

Our first implementation of the out-of-core Cholesky factorization uses the out-of-core versions of the symmetric rank-k update and matrix-matrix multiplication. This implementation uses an in-core triangular solve with multiple right hand sides. This implementation is further enhanced by using an efficient storage scheme. These two-tile implementations achieve high performance on the Cray T3E machines. However, we also implement a one-tile out-of-core Cholesky factorization that calls an out-of-core version of the Triangular solve with multiple right hand sides. This version also uses the efficient storage scheme.

Our results show that the one-tile version achieves higher performance on problem sizes that are a multiple of its tile size. However, the two-tile version achieve higher performance when the problem sizes are a multiple of the two-tile tile size. This is natural, and is to be expected with problem sets of this size. In order to adequately compare these two implementations, the problem sets need to be much greater. This would increase the amount of computation done at the peak of each algorithm.

Further analysis of the two algorithms bring rise to a possible hybrid algorithm that uses both implementations. The algorithm could use the two-tile implementation when it performs the best, and an one-tile implementation when it performs the best. This algorithm should then have better performance than either of the two current algorithms. This would only be possible after analysing larger sets of data.

Our results show that it is fairly simple and efficient to implement an out-of-core Cholesky factorization using POOCLAPACK and PLAPACK. These two packages allowed us to implement three out-of-core support routines, which allowed a straight forward implementation of the out-of-core Cholesky factorization. This framework will allow additional algorithms to be implemented in a similar fashion.

The performance of our algorithm can be improved by furthering the use of overlapping I/O with computation. A version of our two-tile implementation was created that does some overlapping of I/O with computation, however these improvements didn't improve the performance of the code significantly. However, additional overlapping of I/O

with computation could improve the performance of our implementations significantly.

Having an out-of-core Cholesky factorization is not all that useful without an out-of-core triangular solve. This routine would be a most useful addition to the out-of-core library we currently have. Adding an out-of-core triangular solve will complete the overall purpose of introducing an out-of-core Cholesky factorization.

In addition to these four linear algebra routines, it would be most useful to have a full complement of out-of-core routines. This would most easily be accomplished by adding out-of-core implementations of building block routines, like the symmetric rank-k update and matrix-matrix multiplication. Once these routines are created, then more complex out-of-core linear algebra routines, like an LU factorization or a QR factorization, could be implemented. The goal is to implement a complete solver package for dense linear algebra routines.

Bibliography

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

[2] Gregory A. Baker. *Implementation of Parallel Processing to Selected Problems in Satellite Geodesy*. PhD thesis, The University of Texas at Austin, 1998.

[3] Jean-Philippe Brunet, Palle Pederson, and S. Lennart Johnsson. Load-balanced LU and QR factor and solve routines for scalable processors with scalable I/O. In *Proceedings of the 17th IMACS World Congress*, Atlanta, Georgia, July 1994.

[4] 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.

[5] Tom Cwik, Robert van de Geijn, and Jean Patterson. The application of parallel computation to integral equation models of electromagnetic scattering. *Journal of the Optical Society of America A*, 11(4):1538–1545, April 1994.

[6] E. F. D'Azevedo and J. J. Dongarra. The design and implementation of the parallel out-of-core scalapack lu, qr, and cholesky factorization routines. LAPACK Working Note 118 CS-97-247, University of Tennessee, Knoxville, Jan. 1997.

[7] L. Demkowicz, A. Karafiat, and J.T. Oden. Solution of elastic scattering problems in linear acoustics using *h-p* boundary element method. *Comp. Meths. Appl. Mech. Engrg*, 101:251–282, 1992.

[8] 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.

[9] 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.

[10] Y. Fu, K. J. Klimkowski, G. J. Rodin, E. Berger, J. C. Browne, J. K. Singer, R. A. van de Geijn, and K. S. Vemaganti. A fast solution method for three-dimensional many-particle problems of linear elasticity. *Int. J. Num. Meth. Engrg.*, 42:1215–1229, 1998.

[11] Po Geng, J. Tinsley Oden, and Robert van de Geijn. Massively parallel computation for acoustical scattering problems using boundary element methods. *Journal of Sound and Vibration*, 191(1):145–165, 1996.

[12] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1994.

[13] Ken Klimkowski and Robert van de Geijn. Anatomy of an out-of-core dense linear solver. In *Proceedings of the International Conference on Parallel Processing 1995*, volume III - Algorithms and Applications, pages 29–33, 1995.

[14] 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.

[15] Wesley C. Reiley and Robert A. van de Geijn. Pooclapack: Parallel out-of-core linear algebra package. Technical Report TR-99-33, The University of Texas at Austin, Dept. of Computer Sciences, November 1999. Submitted to the International Supercomputing Conference.

[16] David S. Scott. Out of core dense solvers on Intel parallel supercomputers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 484–487, 1992.

[17] David S. Scott. Parallel I/O and solving out-of-core systems of linear equations. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 123–130, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[18] Sivan Toledo and Fred G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computation. In *Proceedings of IOPADS '96*, 1996.

[19] Robert van de Geijn. Zen and the art of high performance parallel computing. PLAPACK Tutorial available from `http://www.cs.utexas.edu/users/plapack`.

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