POOCLAPACK: Parallel Out-of-Core Linear Algebra Package

Wesley C. Reiley
Robert A. van de Geijn

Department of Computer Sciences
The University of Texas
Austin, TX 78712
{rvdg,wesley}@cs.utexas.edu

November 8, 1999

Abstract

In this paper parallel implementation of out-of-core Cholesky factorization is used to introduce the Parallel Out-of-Core Linear Algebra Package (POOCLAPACK), a flexible infrastructure for parallel implementation of out-of-core linear algebra operations. POOCLAPACK builds on the Parallel Linear Algebra Package (PLAPACK) for in-core parallel dense linear algebra computation. Despite the extreme simplicity of POOCLAPACK, 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. Preliminary results from the HP Exemplar X-Class that demonstrate the portability of POOCLAPACK are also given.

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, 10]. Even for those applications, much cheaper methods based on multi-pole expansions, fast multipole methods (FMM), have recently become popular [9]. 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.

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].

A more serious effort to add out-of-core capabilities to LAPACK and ScaLAPACK is provided by SOLAR [15], 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 nonscalability of some of the in-core parallel kernels used.

Our own approach is somewhat different. Since we developed the Parallel Linear Algebra Package (PLAPACK) [16] used as a basis for the Parallel Out-of-Core Linear Algebra Package (POOCLAPACK), we have more flexibility to customize both the in-core and the out-of-core algorithms. This in turn allows us to code the out-of-core algorithms in such a way that the I/O of matrices becomes trivial, reducing the amount of code required to port between platforms and improving performance.

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, 12, 3, 13, 14]. Additional references to applications requiring large dense linear solves are given in [5, 7, 10]. Additional references to research using fast summation methods like FMM are given in [9].

This paper is organized as follows: Section 2 introduces algorithms for solving the Cholesky factorization used to later illustrate the use of POOCLAPACK. Section 3 discusses issues regarding the in-core and out-of-core implementation of sequential Cholesky factorization. Section 4 introduces the POOCLAPACK approach to coding parallel out-of-core dense linear algebra algorithms. Performance is reported in Section 5. Concluding remarks and future directions are given in the final section.

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 (parallel) in-core implementation, the left-looking algorithm has known advantages for out-of-core computation. 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 $b \times b$ submatrices. 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

$$A = \left(\begin{array}{c\|c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \tag{1}$$

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

$$A = \left(\begin{array}{c\|c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \left(\begin{array}{c\|c} L_{TL}^T & L_{BL}^T \\ \hline\hline 0 & L_{BR}^T \end{array}\right) = \left(\begin{array}{c\|c} L_{TL} L_{TL}^T & \star \\ \hline\hline L_{BL} L_{TL}^T & L_{BL} L_{BL}^T + L_{BR} L_{BR}^T \end{array}\right) \tag{2}$$

so that

$$\begin{aligned} A_{TL} &= L_{TL} L_{TL}^T & (3) \\ A_{BL} &= L_{BL} L_{TL}^T & (4) \\ A_{BR} &= L_{BR} L_{BR}^T + L_{BL} L_{BL}^T & (5) \end{aligned}$$

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 submatrix $A_{BR}$and to complete the Cholesky factorization, it suffices to compute the factorization of the updated A_{BR}. This motivates the algorithm given in Fig. 1 (right-looking variant).

2.2 Left-looking variant

To derive a *left-looking* variant for computing this factorization, consider again Eqns. (1)–(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

$$A = \left(\begin{array}{c|c||c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right) \quad \text{and} \quad L = \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array}\right) \tag{6}$$

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

$$A = \left(\begin{array}{c|c||c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c||c} L_{00}^T & L_{10}^T & L_{20}^T \\ \hline 0 & L_{11}^T & L_{21}^T \\ \hline\hline 0 & 0 & L_{22}^T \end{array}\right) \tag{7}$$

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. 1 (left-looking algorithm).

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

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

right-looking algorithm	left-looking algorithm
$A_{11} \leftarrow L_{11} = \text{Chol.fact.}(A_{11})$	$A_{11} \leftarrow A_{11} - A_{10}A_{10}^T$
$A_{21} \leftarrow L_{21} = A_{21}L_{11}^{-T}$	$A_{21} \leftarrow A_{21} - A_{20}A_{10}^T$
$A_{22} \leftarrow A_{22} - L_{21}L_{21}^T$	$A_{11} \leftarrow L_{11} = \text{Chol.fact.}(A_{11})$
	$A_{21} \leftarrow L_{21} = A_{21}L_{11}^{-T}$

continue with

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

enddo

Figure 1: Blocked right- and left-looking Cholesky factorization algorithms.

3 Sequential Implementation

3.1 Sequential In-core Implementation

Either of the two algorithms presented in Section 2 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 $\square$`syrk` that implements the symmetric rank-k update tends to achieve higher performance than the matrix-matrix multiply kernel $\square$`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.

3.2 Sequential Out-of-Core Implementation

The left-looking Cholesky factorization is favored for out-of-core implementations. 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 easier 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.

Let us examine in more detail how to implement an out-of-core Cholesky factorization. Partition

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

where L_{00} is $m \times m$ and we assume that L_{*0} have been computed, while the other parts of A have been left untouched. Here A_{11} is of size $t \times t$, which we will later call a *tile* of size t. All data is assumed to exist on disk.

The following steps will advance the computation so that L_{11} and L_{21} have been computed and have overwritten the corresponding blocks of A:

1. Read A_{11} from disk into memory.
2. Update $A_{11} \leftarrow A_{11} - L_{10}L_{10}^T$ where L_{10} is on disk.
3. Update $A_{11} \leftarrow L_{11} = \text{Chol.fact.}(A_{11})$. Since A_{11} is in memory, this requires an in-core Cholesky factorization. As mentioned, typically a right-looking variant is favored for this subproblem.
4. Write L_{11} to disk, leaving a copy in memory.
5. Update $A_{21} \leftarrow (A_{21} - L_{20}L_{10}^T)L_{11}^{-T}$, where A_{21}, L_{20} and L_{10} are on disk and L_{11} is in memory.
6. Flush all memory.

We must give further details on how to perform steps 2 and 5:

Step 2: $A_{11} \leftarrow A_{11} - L_{10}L_{10}^T$: Here A_{11} is in memory, but L_{10} is on disk. At first glance, this appears to require a read of L_{10}, followed by an in-core symmetric rank-k update. This requires $t \times m$ data to be read, after which mt^2 floating point operations are performed to update A_{11}, for a ratio of t floating point operations for every

floating point number read. However, reading L_{10} requires a considerable amount of memory, thereby limiting the size of t, and thus affecting this ratio.

The following approach retains the benefits of the same ratio t of computation to disk accesses, while maximizing the size of t and thus this ratio: Partition $L_{10} = \left(\begin{array}{c|c|c} L_{10}^{(0)} & \cdots & L_{10}^{(k-1)} \end{array} \right)$ where $L_{10}^{(j)}$ has approximately b columns. Notice that

$$A_{11} - L_{10}L_{10}^T = A_{11} - L_{10}^{(0)}L_{10}^{(0)\,T} - \cdots - L_{10}^{(k-1)}L_{10}^{(k-1)\,T}$$

Thus, the following procedure will perform the update of A_{11} For each $L_{10}^{(j)}$, read this submatrix ($t \times b$ items read), and perform an in-core rank-k update (bt^2 floating point operations). Notice that this maintains the ratio of t computations for each item read from disk. However, by picking b relatively small, very little memory is needed for L_{10}, thus allowing t to be chosen to be much larger. The block size b is typically chosen to equal a block-size that maximizes the performance of the in-core symmetric rank-k update.

This "sequence of narrow symmetric rank-k updates" approach to implementating a larger symmetric rank-k update yields an excellent parallel in-core implementation of symmetric rank-k update. Thus, the out-of-core approach fits naturally with a very good in-core algorithm[1].

Step 5: $A_{21} \leftarrow L_{21} = (A_{21} - L_{20}L_{10}^T)L_{11}^{-T}$: Here only L_{11} is in memory. This time, we partition

$$A_{21} = \left(\begin{array}{c} \hline A_{21}^{(0)} \\ \hline \vdots \\ \hline A_{21}^{(M-1)} \\ \hline \end{array} \right) \text{ and } L_{20} = \left(\begin{array}{c} \hline L_{20}^{(0)} \\ \hline \vdots \\ \hline L_{20}^{(M-1)} \\ \hline \end{array} \right)$$

where $A_{21}^{(i)}$ is approximately $t \times t$. Note that each tile $A_{21}^{(i)}$ must be updated by $A_{21}^{(i)} \leftarrow A_{21}^{(i)} - L_{20}^{(i)}L_{10}$ after which $A_{21}^{(i)} \leftarrow L_{21}^{(i)} = A_{21}^{(i)}L_{11}^{-T}$ can be computed. The out-of-core algorithm for this proceeds as follows: for each i,

- Read $A_{21}^{(i)}$.
- Update $A_{21}^{(i)} \leftarrow A_{21}^{(i)} - L_{20}^{(i)}L_{10}^T$ by reading b columns of $L_{20}^{(i)}$ and L_{10} at a time and performing a general rank-k update, much like for the out-of-core symmetric rank-k update described above. The ratio of computation to disk accesses is equally favorable for this operation as it was for the symmetric rank-k update.
- Update $A_{21}^{(i)} \leftarrow L_{21}^{(i)} = A_{21}^{(i)}L_{11}^{-T}$. Notice that all matrices involved in this operation are in memory, since a copy of L_{11} is kept in memory.
- Write $A_{21}^{(i)}$.

This "sequence of narrow rank-k updates" approach to implementating a matrix-matrix multiply yields an excellent parallel in-core implementation of matrix-matrix multiplication [16, 11] and thus the out-of-core approach fits naturally with the a very good in-core algorithm for this operation.

Careful consideration of the complete out-of-core algorithm shows that in addition to two tiles of size $t \times t$ (one for A_{11} and for $A_{21}^{(i)}$) only a small amount of workspace is needed for storing a few blocks of columns of L_{10} and $L_{20}^{(i)}$. Naturally, t is chosen as large as possible, thus improving the ratio of computation to disk accesses.

[1]Unfortunately, the only reference for this is the actually implementation of symmetric rank-k update in the PLAPACK source. Most likely, ScaLAPACK uses a similar approach.

3.3 Overlapping I/O with computation

It is possible to exploit asynchronous I/O operations to overlap computation with I/O operations. We now discuss opportunities and trade-offs:

1. Read $L_{21}^{(0)}$ while A_{11} is being factored to compute L_{11}.
2. Write L_{11} while $A_{21}^{(0)}$ is being read.
3. Read $A_{21}^{(i+1)}$ while $A_{21}^{(i)}$ is being updated: Notice that this requires storage of three tiles in memory (L_{11}, $A_{21}^{(i)}$ and $A_{21}^{(i+1)}$). This affects the tile size t, and thus the ratio between computation and I/O operations.
4. Read $A_{21}^{(i+1)}$ while $A_{21}^{(i)}$ is being written to disk: One could carefully orchestrate the writing of parts of $A_{21}^{(i)}$ with reading parts of $A_{21}^{(i+1)}$ so that no extra memory is required.
5. Read L_{11} while $L_{21}^{(M-1)}$ from the previous iteration is being updated: Since L_{11} from the previous iteration is required for the final update of $A_{21}^{(M-1)}$ to form $L_{21}^{(M-1)}$, this again requires space for a third tile in memory, affecting the tile size t.
6. Read L_{11} while $L_{21}^{(M-1)}$ from the previous iteration is being written.

Notice that all of the above optimizations would yield minimal benefit: the cost of reading and/or writing these tiles is amortized over many computations, and thus comprises only a small percentage of the total execution time. Thus, we do not consider these optimizations worth the added complexity in the code.

It is in the update $A_{11} \leftarrow A_{11} - L_{10}L_{10}^T$ and $A_{21}^{(i)} \leftarrow A_{21}^{(i)} - L_{20}^{(i)}L_{10}^T$ that there is a more profitable opportunity for overlapping:

7. As discussed in Section 3.2, $A_{11} - L_{10}L_{10}^T = A_{11} - L_{10}^{(0)}L_{10}^{(0)\,T} - \cdots - L_{10}^{(k-1)}L_{10}^{(k-1)\,T}$. Notice that while A_{11} is being updated with $A_{11} - L_{10}^{(j)}L_{10}^{(j)\,T}$, the next block of columns $L_{10}^{(j+1)}$ can already be read from disk.
8. Similarly $A_{21}^{(i)}$ is updated using a few columns of $L_{20}^{(i)}$ and L_{10} at a time, and thus the next blocks of columns of these matrices can be read while the current blocks are being used.

Notice that these last optimizations require minimal extra workspace, since only a few extra columns need to be stored. Moreover, reading of these blocks of columns isn't amortized over nearly as much computation, and thus the benefits may be more noticeable.

3.4 Storage considerations

We must briefly discuss 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 submatrix of size $t \times b$, as is encountered in the out-of-core rank-k updates described above, 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 $t \times t$ matrices A_{11} and $A_{21}^{(i)}$ can be read as one contiguous block, as can $L_{10}^{(j)}$ and blocks of columns of $L_{20}^{(i)}$. For this second scheme, the Cholesky factorization views the matrix as a collection of blocks of rows.

4 Parallel Implementation

4.1 PLAPACK

The Parallel Linear Algebra Package (PLAPACK) is a flexible infrastructure for implementing parallel dense linear algebra libraries. 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. The code segments included in this section are typical of PLAPACK code, in-core as well as out-of-core.

4.2 Data Distribution and File Management

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

4.3 Parallel out-of-core Cholesky factorization

We only describe the parallel implementation 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. 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 submatrices. Each block of t rows is passed to the routine as a view of this data.

We briefly describe the different parts of the routine: 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 3.4 (line 1). The algorithm loops over the panels, partitioning the current panel into L_{10} and A_{11} (lines 6–11). An in-core matrix is created to hold A_{11} and that submatrix is read from disk (lines 14–15). Notice that this requires only a local copy from disk to the in-core matrix. A parallel 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_{21} resides on disk (line 18). We describe this routine in more detail below. Once updated, A_{11} is factored by a call to the parallel Cholesky factorization `PLA_Chol` and written to disk (retaining a copy in memory for now) (lines 21–22). 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 $L_{20}^{(i)}$ and $A_{21}^{(i)}$ (lines 24–28). An in-core matrix is created to hold $A_{21}^{(i)}$ and that submatrix is read from disk (lines 30–32). A parallel matrix-matrix multiplication, `POOCLA_Gemm`, updates $A_{21}^{(i)} \leftarrow A_{21}^{(i)} - L_{20}^{(i)}L_{10}^T$. Implementation of this routine is similar to that of `POOCLA_Syrk` (lines 34–36). 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 level-3 BLAS routine `PLA_Trsm` (triangular solve with multiple right-hand-sides) accomplishes this task (lines 38–43).

```
1   int POOCLA_Chol_by_panels( int N, PLA_Obj *A_row_panels )
2   {
3     < declarations >
4
5     size_done = 0;                                  /* number of columns finished */
6     for ( j=0; j<N; j++ ){
7       PLA_Obj_global_length( A_row_panels[ j ], &t );         /* get tile size    */
8
9       /* View current L_10 and A_11 submatrices */
10      PLA_Obj_vert_split_2( A_row_panels[ j ], size_done,  &L_10, &temp );
11      PLA_Obj_vert_split_2( temp,              t,          &A_11, PLA_DUMMY );
12
13      /* Create an in-core matrix into which to copy A_11 */
14      PLA_Matrix_create_conf_to( A_11, &A_11_in );
15      PLA_Copy( A_11, A_11_in );
16
17      /* Update A_11 <- A_11 - L_10 * L_10, A_11 in-core, L_10 out-of-core */
18      POOCLA_Syrk( PLA_LOWER_TRIANG, PLA_NO_TRANS, min_one, L_10, one, A_11_in );
19
20      /* Factor updated in-core A_11 and write out the result */
21      PLA_Chol( PLA_LOWER_TRIANGULAR, A_11_in );
22      PLA_Copy( A_11_in, A_11 );
23
24      /* Loop over A_21^i */
25      for ( i=j+1; i<N; i++ ){
26        /* View current matrices L_20^i and A_21^i */
27        PLA_Obj_vert_split_2( A_row_panels[ i ], size_done, &L_20_1, &temp );
28        PLA_Obj_vert_split_2( temp,              t, &A_21_1, PLA_DUMMY );
29
30        /* Create an in-core matrix into which to copy A_21^i */
31        PLA_Matrix_create_conf_to( A_21_1, &A_21_1_in );
32        PLA_Copy( A_21_1, A_21_1_in );
33
34        /* Update A_21^i <- A_21^i - L_20 * L_10^T */
35        POOCLA_Gemm( PLA_NO_TRANS, PLA_TRANS,
36                     min_one, L_20_1, L_10, one, A_21_1_in );
37
38        /* Update A_21^i <- L_21^i = A_21^i * L_11^-T */
39        PLA_Trsm( PLA_SIDE_RIGHT, PLA_LOWER_TRIANG, PLA_TRANS, PLA_NONUNIT_DIAG,
40                  one, A_11_in, A_21_1_in );
41
42        /* Write out A_21^i */
43        PLA_Copy(A_21_1_in, A_21_1);
44
45        size_done += t;
46      }
47    }
48    < clean up >
49  }
```

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

4.4 Parallel out-of-core 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, $C \leftarrow \alpha AA^T + \beta C$. A parallel implementation of this operation using POOCLAPACK is given in Fig. 3.

Matrices A and C are passed in as views `A_ooc` and `C`, where `A_ooc` references a matrix stored on disk, while `C` references a matrix stored in-core (line 1). The algorithm starts by scaling $C \leftarrow \beta C$ (line 7). Next, the algorithm loops over blocks of columns, partitioning off the current block $A^{(j)}$ as `A_ooc_1` (lines 11–16). An in-core matrix is created to hold $A^{(j)}$ and that submatrix is read from disk (lines 17–19). A in-core parallel symmetric rank-k update, `POOCLA_Syrk`, updates $C \leftarrow \alpha A^{(j)} A^{(j)\,T} + C$ where $A^{(j)}$ is in-core, referenced by `A_in_1` (line 21). An asynchronous version is given in Fig. 4.

4.5 Parallel out-of-core matrix-matrix multiplication

The Parallel out-of-core matrix-matrix multiply used to update $A_{21}^{(i)} \leftarrow A_{21}^{(i)} - L_{20}^{(i)} L_{10}^T$ is implemented similarly.

5 Performance

In this section, we report preliminary performance achieved with the described PLAPACK based parallel out-of-core implementations of the Cholesky factorization.

5.1 Target Architectures

We demonstrate performance on two different platforms: the Cray T3E-600 (300 MHz) and the HP Exemplar X-Class, 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 MFLOP/s/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 and HP Exemplar X-Class in MFLOP/s:

n	Cray T3E	HP Exemplar
500	418	398
1000	443	496
1500	425	497

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[2]. 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.

The Exemplar we used for our experiments is physically located at Caltech and is part of the National Partnership for Advanced Computational Infrastructure (NPACI). The Exemplar is a Cache-Coherent, Non-Uniform Memory

[2] the `/tmp` directory.

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

Figure 3: POOCLAPACK symmetric rank-k update routine. Matrix A, passed in as object `a_ooc`, is assumed to be stored on disk, while matrix C, passed in as object `C`, is assumed to be in-core. This version does not attempt to overlap I/O with computation.

```
13    while ( TRUE ){
14      /* Check if part of A_ooc yet to be used is of width 0 */
15      PLA_Obj_global_width(  A_ooc_L, &size );
16      if ( ( size = min( size, nb_ooc ) ) != 0 ) {
17        /* view next A^j and asynchronously read to in-core matrix */
18        PLA_Obj_vert_split_2( A_ooc_L, size, &A_ooc_2, &A_ooc_L );
19        PLA_Matrix_create_conf_to( A_ooc_2, &A_in_2 );
20        PLA_Copy_async( A_ooc_2, A_in_2 );
21      }
22      else {
23        if ( last_time )  break;
24        else last_time = TRUE;
25      }
26      if ( !first_time ){
27        PLA_Copy_wait( A_out_1);
28        PLA_Syrk( uplo, transa, alpha, A_in_1, one, C );
29      }
30      else first_time = FALSE;
31      PLA_Obj_view_swap( &A_in_1, &A_in_2 );
32      PLA_Obj_view_swap( &A_ooc_1, &A_ooc_2 );
33    }
```

Figure 4: Asynchronous implementation of the POOCLAPACK symmetric rank-k update routine.

Architecture (CC-NUMA). It consists of a number of shared-memory hypernodes with 16 processors each. Programs can use either shared-memory directives or message-passing libraries (MPI) to access memory on other nodes across the Coherent Toroidal Interconnect (CTI) channels. POOCLAPACK views this machine as a pure distributed memory architecture programmed using MPI. We have less experience with the Exemplar and did not experiment with different I/O options: all I/O on that machine was performed with vanilla UNIX I/O calls. Also, parameters like the distribution, algorithm and tile sizes were optimized for the Cray T3E and simply used for the Exemplar without further attempts at optimization for that machine. The file system used on the Exemplar stripes files across 12 disks. Thus performance numbers for the Exemplar are very preliminary at best.

5.2 Implementations tested

We report performance for five 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 $A_{21}^{(i)} \leftarrow A_{21}^{(i)} - L_{10}^{(i)}L_{10}^T$.

`POOCLA_Chol_by_panels`: This version is given in Fig. 2 and views the matrix as a collection of row panels, as described in Section 3.4. No effort is made to overlap I/O with computation.

`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 $A_{21}^{(i)} \leftarrow A_{21}^{(i)} - L_{10}^{(i)}L_{10}^T$.

The asynchronous versions were not used on the Exemplar.

5.3 Results on the T3E

Performance attained on the Cray T3E-600 is reported in Table 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. The block size described in Section 3.2 used for partitioning L_{10} and $L_{20}^{(i)}$, b, was taken to equal the algorithmic block size.

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 3.4 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.

Algorithm	p	tile size t	1×1 tiles ($n = t$)			2×2 tiles ($n = 2t$)			3×3 tiles ($n = 3t$)		
			MFLOP/s	Time (sec)		MFLOP/s	Time (sec)		MFLOP/s	Time (sec)	
			/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 1: Performance of the various Cholesky factorization routines on the Cray T3E-600.

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 MFLOP/s per processor on 64 processors for the $(3t) \times (3t)$ problem.

5.4 Results on the Exemplar

Great care should be taken when comparing the results collected on the Exemplar and reported in Table 2 with those given for the Cray T3E: For the Exemplar:

- We made no real attempt to optimize either the in-core or out-of-core implementations. In particular, we believe that performance on 8 and 16 processors is less impressive due to a specific detail in the implementation of a PLAPACK kernel that computes a local contribution to a symmetric rank-k update. By recoding this detail, memory conflicts within a hypernode can be reduced, improving performance of this kernel.

- Parameters that optimized execution on the Cray were simply adopted for the Exemplar. In particular the tile size t which determines the amount of in-core memory that was used was considerably smaller than it could be. Recall that t influences the ratio between communication and useful computation. One reason for limiting t was that there was limited disk space available for our experiments.

- We made no attempt to optimize I/O. Indeed, straight-forward Unix calls were used by the I/O related subroutines. Furthermore, we did not implement asynchonous I/O for the Exemplar.

Algorithm	p	tile size t	1×1 tiles ($n = t$)			2×2 tiles ($n = 2t$)			3×3 tiles ($n = 3t$)		
			MFLOP/s	Time (sec)		MFLOP/s	Time (sec)		MFLOP/s	Time (sec)	
			/proc.	Total	I/O	/proc.	Total	I/O	/proc.	Total	I/O
`Chol`	1	2088	179	17.0	4.7	213	114	40	243	337	109
`Chol_by_panel`	1	2088	194	15.6	3.8	299	84	11	333	246	17
`Chol`	4	4704	257	33.8	4.5	262	265	74			
`Chol_by_panel`	4	4704	259	33.5	4.4	333	208	20	354	662	72
`Chol_by_panel`	8	6144	249	38.9	4.3	273	283	46	288	907	146
`Chol_by_panel`	16	8448	173	72.6	19.44	221	455	135	245	1373	359

Table 2: Performance of the various Cholesky factorization routines on the HP Exemplar X-Class. (Preliminary!)

Nonetheless, the performance numbers look encouraging. We intend to further optimize for this architecture in the near future.

6 Conclusion

We have described a simple extension to the PLAPACK parallel linear algebra infrastructure that allows for elegant implementation of out-of-core dense linear algebra algorithms. High performance is reported on the Cray T3E-600.

For now, we have concentrated on the use of existing in-core kernels provided by PLAPACK. However, since both PLAPACK and its out-of-core extension provide a simple abstract programming interface, the implementations lend themselves to customization to attain even higher performance. For example, it is possible to implement an out-of-core Cholesky factorization that requires only one tile to be stored in-core by implementing an out-of-core triangular solve with multiple right-hand-sides. We are in the process of implementing such customizations, which promise even better performance than reported in this paper.

More information

For more information on PLAPACK and POOCLAPACK visit

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

Acknowledgments

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.

A special thanks to Heidi Lorenz-Wirzba at Caltech for helping us obtain the performance results for the NPACI HP Exemplar.

References

[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] 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.

[10] 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.

[11] John Gunnels, Calvin Lin, Greg Morrow, and Robert van de Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98)*, pages 110–116, 1998.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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