Parallel Cholesky Factorization of a Block Tridiagonal Matrix

Thuan D. Cao
John F. Hall
Department of Civil Engineering
California Institute of Technology
Pasadena, CA 91125
tdcao@its.caltech.edu

Robert A. van de Geijn
Department of Computer Sciences
The University of Texas
Austin, TX 78712
rvdg@cs.utexas.edu

April 17, 2002

Abstract

In this paper we discuss the parallel implementation of the Cholesky factorization of a positive definite symmetric matrix when that matrix is block tridiagonal. While parallel implementations for this problem, and closely related problems like the factorization of banded matrices, have been previously reported in the literature, those implementations dealt with the special cases where the block size (bandwidth) was either very large (wide) or very small (narrow). We present a solution that can be used for the entire spectrum of cases, ranging from extremely large (wide) to very small (narrow). Preliminary performance results collected on a Cray T3E-600 distributed memory supercomputer show that our implementation attains respectable performance. Indeed, factorization of a matrix with block size $b = 1000$ and a total dimension of more than $500,000$ takes about 3.6 minutes on 128 processors.

1 Introduction

Dense banded and block tridiagonal matrices are frequently encountered in engineering applications. For example, when modeling structures in civil engineering, the stiffness matrices of two- or three-dimensional structures that are long in one dimension relative to the other dimension(s) have this special structure. Often, these matrices have the additional properties that they are symmetric positive definite (SPD) and ill-conditioned. For large structures, the combination of the size of the block and the total size of the matrix, the number of degrees of freedom (DOF), is such that the memory and/or computational requirements exceed those of a sequential computer. As a result, substructuring methods combined with a solution of the resulting linear system based on a parallel implementation of a block tridiagonal matrix are a viable option for solving these types of problems [4, 13, 16].

Bridge structures, when the span of the bridge is much greater than the height and width, are examples of a problem that fits the above description When this structure is discretized, the DOF will be distributed most densely near the end of each span in the bridge, where the piers, piles, and foundation are. The

resulting matrix problem is quite complex: The nonzero blocks that make up the block tridiagonal matrix in the matrix come from relatively dense coupling within a "slice" of the bridge and between neighboring slices, where these slices consist of degrees of freedom in cross-sections that are orthogonal to the direction of the span. Depending on the density of the discretization points and whether the slice includes part of a pier, pile, and/or foundation, a slice may have a varying number of discretization points, which translates to varying sized blocks on and off the diagonal. The requirement for a solver for this problem go beyond those available in existing sequential and parallel block tridiagonal and banded solvers.

While the parallel implementation of the solution of block tridiagonal matrices, as well that of banded matrices, has been extensively studied [1, 14, 12, 8, 6, 7, 2, 15, 18] we make a number of contributions in this paper: (1) We give a thorough yet simple explanation of the basic techniques for extracting parallelism; (2) We clearly describe the traditional approaches for the extreme cases where the sizes of the blocks are relatively large and relatively small; (3) We present a new hybrid approach for the case where the sizes of the blocks are in an intermediate range; (4) We demonstrate that implementation of these algorithms is relatively straight forward with the Parallel Linear Algebra Package (PLAPACK) [17] infrastructure for implementing dense linear algebra operations; and (5) Performance data collected on a Cray T3E supercomputer demonstrates that respectable performance can be attained on distributed memory architectures. In this paper we only discuss the factorization of the matrix when the matrix has uniform sized blocks. The techniques can be easily extended to the forward and backward substitution processes, which then combined with the factorization yield the solution of the linear system. It can also be extended to the case where the blocks are highly non-uniform, although in that case the problem of load-balancing will need to be addressed.

This paper is organized as follows: In Section 2 we discuss basic techniques for factoring a block tridiagonal SPD matrix. In Section 3 we discuss how these basic techniques yield parallel implementations for distributed memory architectures. The resulting methods apply to the extreme cases where the block size is large or small. In Section 3.3 we show that by combining the techniques for large and small block sizes, we obtain a hybrid approach that encompasses the spectrum of block sizes from very large to very small. Next, in Section 4 we briefly outline the implementation of the hybrid approach using PLAPACK. Performance results are given in Section 5, followed by concluding remarks in the final section.

2 Basic Techniques

2.1 Factorization of A

Consider the block tridiagonal SPD matrix, A,

$$A = \left(\begin{array}{c|c|c|c|c} D_1 & \star & & & \\ \hline E_1 & D_2 & \star & & \\ \hline & E_2 & \ddots & \ddots & \\ \hline & & \ddots & D_{N-1} & \star \\ \hline & & & E_{N-1} & D_N \end{array}\right),$$

where the $\star$s indicate the nonzero symmetric parts of the tridiagonal matrix, and its Cholesky factor, L,

$$L = \left(\begin{array}{c|c|c|c|c} \hat{D}_1 & & & & \\ \hline \hat{E}_1 & \hat{D}_2 & & & \\ \hline & \hat{E}_2 & \ddots & & \\ \hline & & \ddots & \hat{D}_{N-1} & \\ \hline & & & \hat{E}_{N-1} & \hat{D}_N \end{array} \right).$$

Here $\hat{D}_i$, $i = 1, \ldots, N$, are all lower triangular and

$$\begin{array}{rcl} D_1 & = & \hat{D}_1 \hat{D}_1^T \\ \hat{E}_i & = & E_i \hat{D}_i^{-T}, i = 1, \ldots, N-1 \\ D_i - \hat{E}_i \hat{E}_i^T & = & \hat{D}_i \hat{D}_i^T, i = 2, \ldots, N. \end{array}$$

Thus an algorithm for overwriting the block tridiagonal matrix with its Cholesky factor is given by

Algorithm 1 Block tridiagonal Cholesky factorization

do $i = 1, \ldots, N-1$	
$D_i \leftarrow \hat{D}_i = \text{Chol}(D_i)$	$\frac{b^3}{3}$
$E_i \leftarrow \hat{E}_i = E_i \hat{D}_i^{-T}$	b^3
$D_{i+1} \leftarrow D_{i+1} - \hat{E}_i \hat{E}_i^T$	b^3
enddo	
$D_N \leftarrow \hat{D}_N = \text{Chol}(D_N)$	$\frac{b^3}{3}$

Here the cost, in floating point operations (flops), of each step is given to the right of each operation where it is assumed that all nonzero blocks in the matrix are $b \times b$. The total cost of this algorithm in flops is thus given by

$$\frac{1}{3} N b^3 + 2(N-1) b^3 \approx \frac{7}{3} n b^2$$

where $n = Nb$ equals the overall dimension of the matrix.

2.2 Factorization of PAP^T

What should be immediately obvious to the reader is that there appears to be an inherent order in the computations given in Section 2.1. We now show how this dependency can be broken to expose parallelism.

2.2.1 Factorization of PAP^T

Consider the partitioned block tridiagonal matrix in Fig. 1. Let us permute this matrix as in Fig. 2. Then the Cholesky factor of this permuted matrix, L where $PAP^T = LL^T$, is given in Fig. 3. The operations to be performed are

$$
A = \left(\begin{array}{c|c|c|c|c|c}
D_{11} \star & & & & & \\
E_{11} \ddots \ddots & & & & & \\
\quad \ddots D_{N_1 1} & \star & & & & \\
\hline
F_1 & A_2 \star & & & & \\
\hline
 & B_2 & D_{12} \star & & & \\
 & & E_{12} \ddots \ddots & & & \\
 & & \quad \ddots D_{N_2 2} & \star & & \\
\hline
 & & F_2 & A_3 \ddots & & \\
\hline
 & & & \ddots & & \\
 & & & & \ddots & \\
 & & & & \star & \\
\hline
 & & & & F_{p-1} & A_p \star \\
\hline
 & & & & & B_p \; D_{1p} \star \\
 & & & & & E_{1p} \ddots \ddots \\
 & & & & & \quad \ddots D_{N_p p}
\end{array}\right).
$$

Figure 1: Partitioned matrix A.

$$
PAP^T = \left(\begin{array}{c|c|c|c|c|c|c|c}
D_{11} \star & & & & & & & \\
E_{11} \ddots \ddots & & & & & & & \\
\quad \ddots D_{N_1 1} & & & & & \star & & \\
\hline
 & D_{12} \star & & & & \star & & \\
 & E_{12} \ddots \ddots & & & & & \star & \\
 & \quad \ddots D_{N_2 2} & & & & & \star & \ddots \\
\hline
 & & \ddots & & & & & \star \\
\hline
 & & & & D_{1p} \star & & & \star \\
 & & & & E_{1p} \ddots \ddots & & & \\
 & & & & \quad \ddots D_{N_p p} & & & \\
\hline\hline
F_1 & B_2^T & & & & A_2 & & \\
\hline
 & F_2 & B_3^T & & & & A_3 & \\
\hline
 & & \ddots & & & & & \ddots \\
\hline
 & & & F_{p-1} & B_p^T & & & A_p
\end{array}\right)
$$

Figure 2: Permuted matrix A.

- Factor the interior problems (the block tridiagonal matrices in the upper-left quadrant demarked by the triple lines) at a cost of $\sum_{i=1}^{p} \left(\frac{1}{3} N_i + 2(N_i - 1) \right) b^3$ flops.
- Compute $\hat{F}_i$, $i = 1, \ldots, p-1$ at a cost of $(p-1)b^3$ flops.
- Compute $\hat{B}_i$, $i = 2, \ldots, p$ at a cost of $\sum_{i=2}^{p} (N_i + 2(N_i - 1))b^3$ flops.
- Compute the "interface matrix"

$$\left(\begin{array}{c|c|c|c} A_2 - \hat{F}_1 \hat{F}_1^T - \hat{B}_2^T \hat{B}_2 & \star & & \\ \hline \hat{F}_2 \hat{B}_{N_2 2} & A_3 - \hat{F}_2 \hat{F}_2^T - \hat{B}_3^T \hat{B}_3 & \ddots & \\ \hline & \ddots & \ddots & \star \\ \hline & & \hat{F}_p \hat{B}_{N_p p} & A_p - \hat{F}_{p-1} \hat{F}_{p-1}^T - \hat{B}_p^T \hat{B}_p \end{array}\right) \tag{1}$$

  at a cost of $\left(\sum_{i=2}^{p} (N_i + 1) + 2(p-2)\right) b^3$ flops.
- Factor the interface matrix at a cost of $\frac{1}{3}(p-1)b^3 + 2(p-2)b^3$ flops.

the total cost is approximately $(\frac{7}{3} + 4)nb^2$ flops, since $\sum_{i=1}^{p} N_i + (p-1) = N$, the total number of blocks on the diagonal, and $bN = n$, the total matrix dimension. Thus, due to fill-in, the cost of factoring the permuted matrix is almost three times the cost of factoring the block tridiagonal matrix. However, as we will see later, there is now the potential to effectively use p processors for the factorization.

3 Parallel Algorithms

3.1 Factorization of A

When each of the blocks is relatively large, parallelism can be obtained by distributing the matrix using a relatively small distribution block size and using parallel implementations for each of the operations in Alg. 1.

3.2 Factorization of PAP^T

Let us assume that all nonzero blocks on or below the diagonal between the double lines in Fig. 1 are assigned to the same processor. Then the ith processor will hold

$$\left(\begin{array}{ccc|c|c} D_{1i} & \star & & \star & \\ E_{1i} & \ddots & \ddots & & \\ & \ddots & D_{N_i i} & & \star \\ \hline\hline B_i^T & & & A_i & \\ \hline & & F_i & & \end{array}\right)$$

$$L = \text{Chol}(PAP^T) = \left(\begin{array}{c|c|c|c|c|c|c} \hat{D}_{11} & & & & & & \\ \hat{E}_{11} \ddots & & & & & & \\ \ddots \hat{D}_{N_1 1} & & & & & & \\ \hline & \hat{D}_{12} & & & & & \\ & \hat{E}_{12} \ddots & & & & & \\ & \ddots \hat{D}_{N_2 2} & & & & & \\ \hline & & \ddots & & & & \\ \hline & & & \hat{D}_{1p} & & & \\ & & & \hat{E}_{1p} \ddots & & & \\ & & & \ddots \hat{D}_{N_p p} & & & \\ \hline\hline \hat{F}_1 & \hat{B}_{12}^T \cdots \hat{B}_{N_2 2}^T & & & \hat{A}_2 & & \\ \hline & \hat{F}_2 & \hat{B}_{13}^T \cdots \hat{B}_{N_3 3}^T & & \hat{H}_2 & \hat{A}_3 & \\ \hline & & \ddots & & & \ddots & \ddots \\ \hline & & \hat{F}_{p-1} & \hat{B}_{1p}^T \cdots \hat{B}_{N_p p}^T & & \hat{H}_{p-1} & \hat{A}_p \end{array}\right)$$

Here

$$\left(\begin{array}{ccc} \hat{D}_{1i} & & \\ \hat{E}_{1i} & \ddots & \\ & \ddots & \hat{D}_{N_i i} \end{array}\right) = \text{Chol} \left(\begin{array}{ccc} D_{1i} & \star & \\ E_{1i} & \ddots & \ddots \\ & \ddots & D_{N_i i} \end{array}\right), \hat{F}_i = F_i \hat{D}_{N_i i}^{-T}, \left(\begin{array}{ccc} \hat{D}_{1i} & & \\ \hat{E}_{1i} & \ddots & \\ & \ddots & \hat{D}_{N_i i} \end{array}\right) \left(\begin{array}{c} \hat{B}_{1i} \\ \hat{B}_{2i} \\ \vdots \\ \hat{B}_{N_i i} \end{array}\right) = \left(\begin{array}{c} B_i \\ 0 \\ \vdots \\ 0 \end{array}\right),$$

and

$$\left(\begin{array}{c|c|c|c} \hat{A}_2 & & & \\ \hline \hat{H}_2 & \hat{A}_3 & & \\ \hline & \ddots & \ddots & \\ \hline & & \hat{H}_{p-1} & \hat{A}_p \end{array}\right) = \text{Chol} \left(\begin{array}{c|c|c|c} A_2 - \hat{F}_1 \hat{F}_1^T - \hat{B}_2^T \hat{B}_2 & \star & & \\ \hline \hat{F}_2 \hat{B}_{N_2 2} & A_3 - \hat{F}_2 \hat{F}_2^T - \hat{B}_3^T \hat{B}_3 & \ddots & \\ \hline & \ddots & \ddots & \star \\ \hline & & \hat{F}_p \hat{B}_{N_p p} & A_p - \hat{F}_{p-1} \hat{F}_{p-1}^T - \hat{B}_p^T \hat{B}_p \end{array}\right)$$

where $\hat{B}_i^T = \left(\begin{array}{cccc} \hat{B}_{1i}^T & \hat{B}_{2i}^T & \cdots & \hat{B}_{N_i i}^T \end{array} \right)$.

Figure 3: Cholesky factor of permuted matrix.

Notice that all of the following results can be computed in parallel on all processors $i = 1, \ldots, p$ without requiring any communication:

$$\left(\begin{array}{ccc|c|c} \hat{D}_{1i} & & & & \star \\ \hat{E}_{1i} & \ddots & & & \\ & \ddots & \hat{D}_{N_i i} & \star & \\ \hline\hline \hat{B}_{1i}^T & \cdots & \hat{B}_{N_i i}^T & A_i - \hat{B}_i^T \hat{B}_i & \\ \hline & & \hat{F}_i & \hat{F}_i \hat{B}_{N_i i} & -\hat{F}_i \hat{F}_i^T \end{array}\right)$$

To complete the factorization, communication between processors must occur in order to form the entire interface matrix given in (1). after which this smaller block tridiagonal matrix must itself be factored.

We discuss four different approaches to handling the interface matrix.

3.2.1 Approach 1: Redundant sequential

If the size of the blocks and the number of processors are sufficiently small, the amount of computation and data associated with the interface may be sufficiently small that the interface can be duplicated to all processors and redundantly factored. For this we suggest the following approach:

- On each processor create a block tridiagonal matrix to hold the interface, which is initialized to zero.
- On each processor fill in the contributions to the interface matrix. For example, processor 2 contributes

$$\left(\begin{array}{c|c|c|c|c} A_2 - \hat{B}_2^T \hat{B}_2 & \star & & & \\ \hline \hat{F}_2 \hat{B}_{N_2 2} & -\hat{F}_2 \hat{F}_2^T & 0 & & \\ \hline & 0 & 0 & \cdots & \\ \hline & & \ddots & \ddots & 0 \\ \hline & & & 0 & 0 \end{array}\right)$$

- Sum the contributions from all processors together to assemble the interface matrix, leaving the result duplicated on all processors.
- Redundantly factor the interface matrix on each processor.

3.2.2 Approach 2: Distributed sequential

Notice that in order to compute $\hat{A}_i$ and $\hat{H}_i$ on processor i, that processor must receive $\hat{F}_{i-1}\hat{F}_{i-1}^T + \hat{H}_{i-1}\hat{H}_{i-1}^T$ from processor $i-1$ to be subtracted from $A_i - B_i^T B_i$. Thus, the factorization of the interface can happen distributed among the processors, passing $\hat{F}_{i-1}\hat{F}_{i-1}^T + \hat{H}_{i-1}\hat{H}_{i-1}^T$ from processor $i-1$ to processor i once this update has been computed. Notice that the computational cost of factoring the interface is as if it were performed on one processor, due to the inherent dependencies. One benefit of this approach is that the interface matrix needs not be duplicated, which reduces memory requirements.

3.2.3 Approach 3: Distributed parallel

A third way is to assemble the interface matrix into a distributed matrix after which the approach presented in Section 3.1 can be applied. By using a sufficiently small distribution block size, some parallelism can be attained during the factorization of the interface. The approach also has the benefit that a full copy of the interface needs not be duplicated on any or all nodes.

Unfortunately, the communication used to assemble the matrix as well as the cost of the parallel factorization of the interface matrix is difficult to model. We will present empirical data in Section 5.

3.2.4 Approach 4: Cyclic reduction

A standard approach for solving the interface problem is to use cyclic reduction. We will not consider this since it is messy to implement and our alternative, simpler, approaches yield high performance.

3.3 Hybrid

When the blocks are of intermediate size, only a few blocks will fit in the local memory of an individual processor. As a result, most of the computation is in the interface which is the part that does not parallelize well. The solution is to use any of the approaches in Sections 3.2 except that a group is assigned to the task previously performed by a single processor. Within each group the local computation is then parallelized by cooperating as described in Section 3.1

3.3.1 Approach 1: Redundant by groups

If the size of the blocks and the number of groups are still sufficiently small, the amount of computation and data associated with the interface may be sufficiently small that the interface can be duplicated to each group of processors and redundantly factored by each group. For this we suggest to modify the approach in Section 3.2.1 as follows:

- Within each group create a distributed block tridiagonal matrix to hold the interface, which is initialized to zero.
- Within each group fill in the contributions to the interface matrix.
- By making the contributions to the interface distributed identically, the part of the distributed matrix can be added by reducing it among corresponding processors from each group.
- Redundantly factor the interface matrix within each group.

3.3.2 Approach 2: Distributed sequential (between groups)

A similar modification can be made to Approach 2 in Section 3.2.2.

3.3.3 Approach 3: Distributed parallel

If the interface matrix is very large, Approach 3 in Section 3.2.3 can be modified by assembling the interface matrix distributed among all processors.

3.3.4 Approach 4: Cyclic reduction

An approach that uses cyclic reduction to solve the interface problem using groups can be similarly derived from the approach in Section 3.2.4.

4 Implementation

An astute reader will have already noticed that implementations of the hybrid approaches in Section 3.3 trivially encompass all the implementations discussed in Sections 3.1 and 3.2. After all, by letting all processors be part of the same group, any of the approaches in Section 3.3 becomes the one given in Section 3.1 while assigning only one processor to each group yields the methods in Section 3.2. Thus we implemented the hybrid algorithms.

Notice that the factorization of a block tridiagonal matrix requires three operations: $A \leftarrow L = \text{Chol}(A)$, $B \leftarrow BL^{-T}$, and $A \leftarrow A - LL^{-T}$. The first is a Cholesky factorization, the second is a triangular solve with multiple right-hand-sides, while the last is a symmetric rank-k update. These last two operations are part of the level-3 Basic Linear Algebra Subprograms (BLAS) interface [10]. The factorization of the permuted matrix requires, in addition, a matrix-matrix multiplication, which is also part of the level-3 BLAS. Thus, given parallel implementations for these operations it is straight-forward to implement the block tridiagonal factorization in Section 3.1. Given the observations in Section 3.3, except for assembling the block tridiagonal interface matrix to a group or all processors and we have all components for a hybrid block tridiagonal factorization.

Our implementations are based on PLAPACK, which is an infrastructure for coding parallel linear algebra libraries. In addition to providing a facility for distributing matrices to processors, it also provides a set of dense linear algebra operations, including a dense Cholesky factorization and the Basic Linear Algebra Subprograms. The implementations of these individual operations are scalable and high-performing.

We thus concentrate on brief descriptions of how the interface matrices are assembled.

4.1 Approach 1: Redundant by groups

In this first approach, we limit ourselves to using groups with equal numbers of processors. Observe that if we assign the complete interface matrix redundantly to each group and corresponding processors from each group hold corresponding parts of the interface matrix, then each group can fill its copy of the interface matrix with its contribution to the interface, after which a call to MPI_Allreduce among corresponding processors can be used to leave a copy of the assembled interface distributed among each group.

4.2 Approach 2: Distributed sequential (between groups)

We have not yet implemented this approach.

4.3 Approach 3: Distributed parallel

For this approach, the interface is distributed among all processors, as a tridiagonal blocked matrix would as described in Section 3.1. PLAPACK provides an application interface (PLA/AI) that allows individual

processors to contribute to a globally distributed matrix. This interface takes a contribution, which is typically a submatrix of the global matrix, and performs all communication required to add this contribution to the global matrix. More precisely, the interface uses MPI and a polling mechanism to simulate one-sided communication. For details, see [17].

4.4 Approach 4: Cyclic reduction

We have not yet implemented this approach.

5 Preliminary Performance Results

In this section, we report performance attained on a Cray T3E-600 supercomputer. All results are for the case where 64-bit arithmetic is performed as part of the computation. In most graphs, we report MFLOPS/sec/proc. (millions of floating point operations per second per processor).

In Fig. 4 we demonstrate the benefits of the hybrid method relative to the traditional wide block ($p = g$) and small block ($g = 1$) methods. In the graph we show the time required to factor a matrix with block size $b = 500$, increasing the total matrix size so that the ratio of blocks to processors is kept approximately constant. More precisely, for the hybrid algorithm, the number of interior blocks per group of four is equal to sixteen. For this size of block, the small block method is simply not practical, since the interface matrix quickly becomes very large. Thus we only report data for the hybrid and large block methods. Observe that while the large block method exhibits limited parallelism, leading to the total time for the factorization increasing as the problem size and number of processors increases, this is not the case for the hybrid method.

In Fig. 5 we report the performance of the PLAPACK routines that are called as part of our hybrid factorization method. While we report performance for only four processors, in other papers we have demonstrated that these PLAPACK implementations are scalable in the sense that if memory utilization per processor is scaled, then performance per processor is roughly maintained [5, 11, 3, 9]. In our setting, this means that if the block size b grows with $\sqrt{g}$, where g is the number of nodes in a group, then the attained MFLOPS/sec/proc. for each of these operation is roughly maintained.

In Fig. 6 we report performance as a function of the block size b when g and p are fixed. In that figure, we show the difference in performance between Approaches 1 and 3 in Section 4, labeled by "PLA/Reduce" and "PLA/AI", respective. Notice that in that plot we use the operation count $\frac{7}{3}nb^2$ to compute the rate of computation since this represents the *useful* flops being performed. The performance reported is that attained once the number of interior blocks is reasonably large. Notice that the "PLA/Reduce" curve drops off at $b = 700$. This is due to the fact that the interface matrix must be stored within each group and therefore the number of interior points becomes limited due to memory constraints. Indeed, the problem doesn't fit after the block size increases beyond $b = 700$. When Approach 3 is used, distributing the interface matrix to all processors, much large block sizes can be accomodated.

In Fig. 7 we show where time is spent during the factorization for Approach 3. In that figure, we show the time for the local computation, the assembly of the interface using the PLAPACK application interface, and the factorization of the interface, as well as the total time. As can be expected, as the number of