CHAPTER

Cholesky Factorization 15

Section 8.8 used matrix multiplication to introduce the notion of recursive linear algebra and **cache-oblivious** algorithms, based on the **fork–join** pattern. The Cholesky decomposition example here shows how the pattern applies to parallelizing other operations in dense linear algebra, even when the matrices are triangular. The chapter also gives a brief introduction to the art of using Fortran-oriented **BLAS** routines in C/C++. Though the algorithms apply to large matrices, all you have to know is algebra on 2×2 matrices to understand how they work.

15.1 FORTRAN RULES!

The matrix layout used in the Cholesky routine and subsidiary routines is *column-major*, just like Fortran, not as usual in C/C++, so the layout will be compatible with common implementations of the Basic Linear Algebra Subprograms (BLAS) and Linear Algebra Package (LAPACK). This means each *column* of the matrix is packed consecutively in memory, not each row as is usual in C/C++.

The reason for using a BLAS column-major layout is pragmatic:

- Several recursive routines require serial routines for base cases. Efficient implementations of these serial routines are widely available (such as in Intel MKL) for Fortran array layout.
- Using the same layout enables our routines to be substituted for BLAS routines.

In both cases, the equivalence to BLAS routines may require some thin wrappers that reorder arguments, take addresses of arguments, add implied arguments, or resolve overloading. For example, the following wrapper defines a thin wrapper that implements a triangular solver for leaf cases, using Intel MKL:

```
inline void leaf_trsm(int m, int n, const float b[], int
   ldb, float a[], int lda, char side=‘L’, char transa=‘N
   ’, char diag=‘N’, float alpha=1 ) {
   strsm(&side, "Lower", &transa, &diag, &m, &n, &alpha,
      b, &ldb, a, &lda);
}
```

The wrapper demonstrates the convention for passing arrays. Here `a` represents a lower-triangular $m \times n$ matrix A. The value `lda` is the stride in memory between columns, measured in array elements.

The parameter's name is an abbreviation for *leading dimension of A*. So array element $A_{i,j}$ is accessed as `a[i+j*lda]`. Likewise, there is a rectangular matrix B such that $B_{i,j}$ is accessed as `b[i+j*ldb]`. Following Fortran conventions in C/C++ is a painful, but practical, concession to the reality that production BLAS libraries cater to Fortran.

Routine `leaf_trsm` also illustrates the BLAS penchant for using characters where a C/C++ programmer would use an enum. The default values in the wrapper cause it to solve $A \times X = \alpha \cdot B$, and overwrite A with X. Other values change the problem slightly, as follows:

Argument	Mnemonic	Meaning
`side`	Left	Solve $op(A) \times X = \alpha \cdot B$
	Right	Solve $X \times op(A) = \alpha \cdot B$
`transa`	Transpose	*op* is transpose
	Not transpose	*op* is identity
`diag`	Unit	Diagonal elements of *A* are one
	Not unit	Assuming nothing about diagonal elements of *A*

The argument should be the first letter of the mnemonic, in either lower- or uppercase. For example, passing `side='R'` and `transa='T'` to our wrapper asks it to solve $X \times A^T = \alpha \cdot B$. The wrapped BLAS routine expects a *pointer* to the letter. That is why passing a C string for the mnemonic works, too. The second argument to `strsm` could be `"Lollipop"` instead of `"Lower"` to `trsm`, and it would not make any difference.

The examples also use a matrix multiply routine `parallel_gemm`, a parallel generic routine similar to the BLAS routine `sgemm`. It is assumed to be parallelized using the techniques discussed for matrix multiplication in Section 8.8. The usual BLAS version is quite general. It overwrites a matrix C with $\alpha \cdot op(A) \times op(B) + \beta C$, where *op* is either an identity or transpose operation. Our version has only 12 arguments, since $\beta = 1$ in all our examples. An invocation of our routine looks like:

`parallel_gemm`*(m, n, k, a, lda, b, ldb, c, ldc, transa, transb, α);*

where parameter α is optional and defaults to 1. The rest of the parameters specify the three matrices as follows:

Matrix	Dimensions	Base	Stride	Transpose?
op(*A*)	$m \times k$	*a*	*lda*	*transa*
op(*B*)	$k \times n$	*b*	*ldb*	*transb*
C	$m \times n$	*c*	*ldc*	

The transpose arguments follow the conventions described for `leaf_trsm`.

Our brief introduction to BLAS omitted all possible argument options, such as for upper triangular or complex matrices. Since the BLAS are widespread, searching the Internet for `strsms`, `ssyrk`, and `sgemm` will tell you what you need to know about BLAS routines used in this chapter.

15.2 RECURSIVE CHOLESKY DECOMPOSITION

Our Cholesky decomposition factors a symmetric positive-definite matrix A into a lower triangular matrix L such that $A = LL^T$. The recursion is based on treating A and L as 2×2 matrices of submatrices, like this:

$$A = \begin{bmatrix} A_{00} & A_{10}^T \\ A_{10} & A_{11} \end{bmatrix} \quad L = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix}.$$

The 0 represents a zero submatrix. Submatrices L_{00} and L_{11} are triangular. Submatrices A_{00} and A_{11} are symmetric. Because of the symmetry, submatrix A_{10}^T is redundant. To reduce clutter, it will be omitted from diagrams.

The equation $A = LL^T$ can be rewritten as:

$$\begin{bmatrix} A_{00} & \\ A_{10} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix} \begin{bmatrix} L_{00}^T & L_{10}^T \\ 0 & L_{11}^T \end{bmatrix},$$

which yields the following equations:

$$A_{00} = L_{00}L_{00}^T, \tag{15.1}$$

$$A_{10} = L_{10}L_{00}^T, \tag{15.2}$$

$$A_{11} = L_{10}L_{10}^T + L_{11}L_{11}^T. \tag{15.3}$$

A **divide-and-conquer** algorithm is a practical corollary of the equations:

1. Solve Equation 15.1 for L_{00}. This step is a Cholesky decomposition of A_{00}, which can be done recursively, because the fact that A is positive semi-definite implies that A_{00} is positive semi-definite.
2. Solve Equation 15.2 for L_{10}. This step is called a *triangular solve*, because L_{00} is triangular.
3. Solve Equation 15.3 for L_{11} as follows:
   a. Let $A'_{11} = A_{11} - L_{10}L_{10}^T$. The computation can overwrite A_{11} with A'_{11}. This step is called *symmetric rank update*.
   b. Solve $A'_{11} = L_{11}L_{11}^T$ for L_{11}. This step is a Cholesky decomposition of A_{11}.

```
template<typename T>
void parallel_potrf( int n, T a[], int lda ) {
   if( double(n)*n*n<=CUT ) {
      // Leaf case – solve with serial LAPACK
      leaf_potf2( n, a, lda );
   } else {
      int n2=n/2;
      // Solve $A_{00} = L_{00}L_{00}^T$ for $L_{00}$ and set $A_{00} = L_{00}$
      parallel_potrf( n2, a, lda );
      // Solve $A_{10} = L_{10}L_{00}^T$ for $L_{10}$ and set $A_{10} = L_{10}$
      parallel_trsm( n-n2, n2, a, lda, a+n2, lda );
      // Set $A_{11} = A_{11} - L_{10} \times L_{10}^T$
      parallel_syrk( n-n2, n2, a+n2, lda, a+n2+n2*lda, lda );
      // Solve $A_{11}^T = L_{11}L_{11}^T$ for $L_{11}$
      parallel_potrf( n-n2, a+n2+n2*lda, lda );
   }
}
```

LISTING 15.1

Recursive Cholesky decomposition. The inscrutable name `potrf` follows LAPACK conventions [ABB+99].

Listing 15.1 shows code for the algorithm. The algorithm introduces no parallelism. However, it sets up two subproblems where parallelism can be introduced.

15.3 TRIANGULAR SOLVE

The first subproblem that enables parallelism is the triangular solve. It solves for X in the equation $XB^T = A$, where B is a lower triangular matrix. There are two different ways to split the matrices:

1. Split X and A horizontally, so the equation decomposes into:

$$\left[\begin{array}{c} X_0 \\ \hline X_1 \end{array}\right] B^T = \left[\begin{array}{c} A_0 \\ \hline A_1 \end{array}\right],$$

   which yields the equations:

$$X_0 B^T = A_0$$
$$X_1 B^T = A_1.$$

   An opportunity for parallelism is introduced because X_0 and X_1 can be solved for independently.

2. Split X and A horizontally, and B^T on both axes, so the equation decomposes into:

$$\left[\begin{array}{c|c} X_0 & X_1 \end{array}\right] \begin{bmatrix} B_{00}^T & B_{10}^T \\ 0 & B_{11}^T \end{bmatrix} = \left[\begin{array}{c|c} A_0 & A_1 \end{array}\right],$$

which yields the equations:

$$X_0 B_{00}^T = A_0, \tag{15.4}$$

$$X_0 B_{10}^T + X_1 B_{11}^T = A_1. \tag{15.5}$$

No opportunity for parallelism is introduced, because Equation 15.4 must be solved first to find X_0 before Equation 15.5 can be solved to find X_1.

In the latter case, the steps are:

1. Solve the equation $X_0 B_{00}^T = A_0$ for X_0, which is a triangular solve.
2. Set $A_1' = A_1 - X_0 B_{10}^T$. The computation can overwrite A_1' with A'.
3. Solve the equation $X_1 B_{11}^T = A'$ for X_1, which is a triangular solve.

At first, using the second split seems useless. But, if the first split is applied exclusively, then X and A in the leaf cases are long skinny row vectors, and each element of B^T is used exactly once, with no reuse. Consequently, consumption of memory bandwidth will be high. It is better to alternate between splitting vertically and splitting horizontally, so the subproblems remain roughly square and to encourage reuse of elements. Furthermore, the second split is not a complete loss, because the matrix multiplication in step 2 can be parallelized.

Listing 15.2 shows a Cilk Plus incarnation of the algorithm. Translation to TBB is a matter of rewriting the fork–join with `tbb::parallel_invoke`. The number of floating-point arithmetic operations is about `m2`^{2}`n2`/6. The recursion stops when this number is 6· `CUT` or less. The cast to `double` in that calculation ensures that the estimate does not err from overflow.

The recursive decomposition into smaller matrices makes the algorithm into a **cache-oblivious** algorithm (Section 8.8). Like the cache-oblivious matrix multiplication in Section 8.8, one of the recursive splits does not introduce any parallelism. It is beneficial nonetheless because splitting on the longest axis avoids creating long, skinny matrices, which improves cache behavior, as was explained in Section 8.8 for the matrix multiplication example.

15.4 SYMMETRIC RANK UPDATE

The other subproblem with parallelism is the symmetric rank update, which computes $A' = A - CC^T$ and overwrites A with A'. This is similar to routine `MultiplyAdd` in Section 8.8, except that subtraction replaces addition, and only the lower triangular portion needs to be computed, because the result is a symmetric matrix. As with the triangular solve, there are two ways to split the problem, one that enables parallelism and the other that, though serial, helps keep the subproblem roughly "square."

The equations for the two splits are:

1.

$$\begin{bmatrix} A_{00}' & \\ A_{10}' & A_{11}' \end{bmatrix} = \begin{bmatrix} A_{00} & \\ A_{10} & A_{11} \end{bmatrix} - \begin{bmatrix} C_0 \\ \hline C_1 \end{bmatrix} \left[\begin{array}{c|c} C_0^T & C_1^T \end{array} \right]$$

```
template<typename T>
void parallel_trsm( int m, int n, const T b[], int ldb, T a[], int lda ) {
   if( double(m)*m*n<=CUT ) {
      // Leaf case−solve with serial BLAS
      leaf_trsm(m, n, b, ldb, a, lda, ‘R’, ‘T’, ’N’ );
    } else {
      if( m>=n ) {
         // Partition A into $\left[\begin{array}{c} A_0 \\ \hline A_1 \end{array}\right]$
         int m2=m/2;
         // Solve $X_0 \times B^T = A_0$, and set $A_0 = X_0$
         cilk_spawn parallel_trsm( m2, n, b, ldb, a, lda );
         // Solve $X_1 \times B^T = A_1$, and set $A_1 = X_1$
         parallel_trsm( m−m2, n, b, ldb, a+m2, lda );
      } else {
         // Partition B into $\begin{bmatrix} B_{00}^T & B_{10}^T \\ 0 & B_{11}^T \end{bmatrix}$ and A into $\left[\begin{array}{c|c} A_0 & A_1 \end{array}\right]$
         // where $B_{00}$ and $B_{11}$ are lower triangular matrices
         int n2=n/2;
         // Solve $X_0 \times B_{00}^T = A_0$, and set $A_0 = X_0$
         parallel_trsm( m, n2, b, ldb, a, lda );
         // Set $A_1 -= A_0 * B_{10}^T$
         parallel_gemm( m, n−n2, n2,
                      a, lda, b+n2, ldb, a+n2*lda, lda,
                     ‘N’, ‘T’, T(−1), T(1) );
         // Solve $X_1 \times B_{11}^T = A_1$, and set $A_1 = X_1$
         parallel_trsm( m, n−n2, b+n2+n2*ldb, ldb, a+n2*lda, lda );
      }
   }
   // Implicit cilk_sync
}
```

LISTING 15.2

Parallel triangular solve in Cilk Plus.

which yields the equations:

$$A'_{00} = A_{00} - C_0 C_0^T, \tag{15.6}$$

$$A'_{10} = A_{10} - C_1 C_0^T, \tag{15.7}$$

$$A'_{11} = A_{11} - C_1 C_1^T. \tag{15.8}$$

Equation 15.7 is a matrix multiplication, which can be parallelized as discussed in Section 8.8. The other two equations are symmetric rank updates. All three can be computed in parallel.

```
void parallel_syrk( int n, int k, const T c[], int ldc, T a[], int lda ) {
   if( double(n)*n*k<=CUT ) {
      leaf_syrk( n, k, c, ldc, a, lda );
   } else if( n>=k ) {
      int n2=n/2;
      cilk_spawn parallel_syrk( n2, k, c, ldc, a, lda );
      cilk_spawn parallel_gemm( n-n2, n2, k,
                                c+n2, ldc, c, ldc, a+n2, lda,
                                'N', 'T', T(-1), T(1) );
      parallel_syrk( n-n2, k, c+n2, ldc, a+n2+n2*lda, lda );
   } else {
      int k2=k/2;
      parallel_syrk( n, k2, c, ldc, a, lda );
      parallel_syrk( n, k-k2, c+k2*ldc, ldc, a, lda );
   }
   // Implicit cilk_sync
}
```

LISTING 15.3

Parallel symmetric rank update in Cilk Plus.

2.

$$A' = A - \left[C_0 \,\middle|\, C_1 \right] \left[\frac{C_0^T}{C_1^T} \right],$$

which yields the equation:

$$A' = A - C_0 C_0^T - C_1 C_1^T.$$

This is essentially two symmetric rank updates, using C_0 and C_1.

The second split enables parallel computation if temporary storage is allocated to compute $C_1 C_1^T$. However, doing so invites the same space and bandwidth problems as doing so for matrix multiplication, as discussed in Section 8.8, so the code will do the two updates serially.

Listing 15.3 shows the Cilk Plus code. Translation to TBB is a matter of using `tbb::parallel_invoke` to express the three-way fork, as shown in Listing 15.4.

15.5 WHERE IS THE TIME SPENT?

The Cholesky decomposition as described involves recursive calls among four algorithms, as shown in Figure 15.1. Each algorithm's base case is a serial version of that algorithm. Since the recursive steps do practically no work, but merely choose submatrices, most of the work is done in the serial leaf code. That leaves a question of how the work is apportioned across the three kinds of leaves. It turns out that the leaf matrix multiplications dominate, because they apply to off-diagonal blocks, whereas

```
tbb::parallel_invoke(
   [=]{parallel_syrk( n2, k, c, ldc, a, lda );},
   [=]{parallel_gemm( n−n2, n2, k,
                  c+n2, ldc, c, ldc, a+n2, lda,
                  'N', 'T', T(−1), T(1) );},
   [=]{parallel_syrk( n−n2, k, c+n2, ldc, a+n2+n2*lda, lda );});
```

LISTING 15.4

Converting parallel symmetric rank update from Cilk Plus to TBB. The listing shows the TBB replacement for lines 6 to 10 of Listing 15.3.

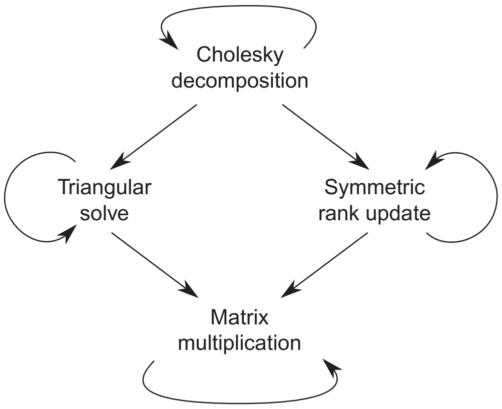

FIGURE 15.1

Cholesky call graph. The call chains eventually reach matrix multiplication, which dominates the running time.

the leaf triangular solve and leaf symmetric rank update apply to diagonal blocks. There are $\Theta(n^2)$ off-diagonal blocks but only $\Theta(n)$ diagonal blocks.

Using fork–join for Cholesky decomposition does have a drawback—it artificially lengthens the span, similar to how recursive decomposition lengthened the span of the binomial lattice problem (Section 8.12.1). A system such as Intel Concurrent Collections avoids lengthening the span by expressing only the dependencies necessary to solve the problem [CKV10]. There is no free lunch, however. That reduction in span discards the simple call-tree reasoning of the Cilk Plus solution and the cache-oblivious benefits, although sufficiently large leaf cases can take care of the memory bandwidth issues. In that case, the concurrent collections approach can deliver superior performance.

15.6 SUMMARY

The Cholesky example demonstrates the power of being able to nest parallelism and think about it locally in each routine. Every routine, though parallel, is called as if it were a serial routine. It also demonstrates how linear algebra can often be done recursively.