Zen and the Art of High Performance Parallel Computing

C Version

by

Greg Morrow
Robert van de Geijn

October 16, 1998

Contents

Chapter 1

Introduction

For the moment, just read

R. van de Geijn, "Using PLAPACK," The MIT Press, 1997[1]

[1] http://www.cs.utexas.edu/users/plapack/using.html

Chapter 2

Hey, I Just Want to Solve My `$#%#$` Problem

Basic assumption: the reader already has used MPI in the past.

2.1 A simple driver

The sample program given in Fig. 2.1–2.2 shows how to initialize MPI and PLAPACK, create a problem to be solved, and call various PLAPACK routines to solve them.

> **Go directly to Section 2.2 if you really just want to know how to distribute your problem to the processors, call PLAPACK, and get the answer back!**

We highlight some of the more interesting parts of the code:

Header files

Our sample routine includes header files for both MPI and PLAPACK.

```
#include "mpi.h"
#include "PLA.h"
```

Declarations

MPI and PLAPACK is object based, which means that many details of groups of processors, matrices and vectors are hidden in *opaque objects* (datastructures to which a program doesn't have direct access). In essence, a matrix or vector is given a handle (pointer to the datastructure). By giving this handle to inquiry routines, information about the matrices or vectors can be extracted.

Declaring a communicator

```
  MPI_Comm comm;
```

```c
#include "mpi.h"
#include "PLA.h"

int main(int argc, char *argv[])
{
  MPI_Comm comm;
  PLA_Template templ = NULL;
  PLA_Obj  A = NULL,           x = NULL,         b = NULL,
           minus_one = NULL,   b_norm = NULL;
  double   *locA, b_norm_value, time;
  int      size, nb_distr, me, nprocs, dummy, info = 0;

  MPI_Init(&argc, &argv);

  PLA_Comm_1D_to_2D_ratio(MPI_COMM_WORLD, 1.0, &comm);
  PLA_Init(comm);

  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  if (0 == me) {
    printf("enter matrix size and distribution block size:\n");
    scanf("%d %d", &size, &nb_distr );
  }

  MPI_Bcast(&size,     1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&nb_distr, 1, MPI_INT, 0, MPI_COMM_WORLD);

  PLA_Temp_create( nb_distr, 0, &templ );

  PLA_Matrix_create(  MPI_DOUBLE, size, size,
                      templ, PLA_ALIGN_FIRST, PLA_ALIGN_FIRST, &A );

  PLA_Mvector_create( MPI_DOUBLE, size, 1, templ, PLA_ALIGN_FIRST, &x );

  PLA_Mvector_create( MPI_DOUBLE, size, 1, templ, PLA_ALIGN_FIRST, &b );

  PLA_Create_constants_conf_to( A, &minus_one, NULL, NULL );

  create_problem( A, x, b );

  MPI_Barrier( MPI_COMM_WORLD );
  time = MPI_Wtime( );

  info = PLA_Chol( PLA_LOWER_TRIANGULAR, A );

  if ( info != 0 ) {
    printf("nonpositive definite matrix detected\n");
  }
```

Figure 2.1: A simple driver routine for testing the Cholesky factorization.

```
  else {
    PLA_Trsv( PLA_LOW_TRIAN, PLA_NO_TRANS, PLA_NONUNIT_DIAG, A, b );
    PLA_Trsv( PLA_LOW_TRIAN, PLA_TRANS,    PLA_NONUNIT_DIAG, A, b );

    MPI_Barrier( MPI_COMM_WORLD );
    time = MPI_Wtime() - time;

    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    if ( me == 0 )
      printf("n = %d, time = %f, MFLOPS/node = %f\n", size, time,
             1.0/3.0 * size * size * size / time * 1.0e-6 / nprocs );

    process_answer( b );

    PLA_Axpy( minus_one, x, b );

    PLA_Mscalar_create( MPI_DOUBLE, PLA_ALL_ROWS, PLA_ALL_COLS,
                        1, 1, templ, &b_norm );

    PLA_Nrm2( b, b_norm );

    if ( me == 0 ) {
      PLA_Obj_get_local_contents( b_norm, PLA_NO_TRANS, &dummy, &dummy,
                                  &b_norm_value, 1, 1 );
      printf( "Norm of difference between original x and computed x: %e \n",
              b_norm_value );
    }
  }

  PLA_Obj_free(&A);             PLA_Obj_free(&x);
  PLA_Obj_free(&b);             PLA_Obj_free(&minus_one);
  PLA_Obj_free(&b_norm);

  PLA_Temp_free(&templ);
  PLA_Finalize( );
  MPI_Finalize( );
}
```

Figure 2.2: A simple driver routine for testing the Cholesky factorization, Part 2.

```
PLA_Template templ = NULL;
PLA_Obj  A = NULL,          x = NULL,         b = NULL,
         minus_one = NULL,   b_norm = NULL;
double   *locA, b_norm_value, time;
int      size, nb_distr, nb_alg, me, nprocs, dummy, info = 0;
```

Initialize MPI

MPI users will recognize the following as a standard MPI initialization call. MPI must be initilized before PLAPACK is initialized.

```
MPI_Init(&argc, &argv);
```

Initialize PLAPACK

PLAPACK requires matrices to be distributed to logical two-dimensional meshed of nodes. To facilitate this, the initialization routine for PLAPACK requires an MPI communication with a two-dimensional topology as an argument. The routine

```
PLA_Comm_1D_to_2D_ratio( MPI_Comm comm_1D, double ratio, MPI_Comm comm_2D )
```

creates a 2D communicator `comm_2D` from 1D communicator `comm_1D` with a ration of rows to columns of approximately `ratio`. Thus, the following code segment initialized PLAPACK with a 2D communicator for a mesh of nodes that is roughly square.

```
PLA_Comm_1D_to_2D_ratio(MPI_COMM_WORLD, 1.0, &comm);
PLA_Init(comm);
```

Get problem parameters

Our sample program will require two inputs from the user: the problem size `size`, and the distribution block size `nb_distr`. At this moment, the reader need not concern him/herself too much with the distribution block size, other than that if it is too big, load-imbalance will result, and if it is too small, communication overhead will increase. We suggest trying distribution block sizes like $8, 16, 32, 50, 64$. After processor with rank 0 reads the parameters, they are broadcast to all processors.

```
MPI_Comm_rank(MPI_COMM_WORLD, &me);
if (0 == me) {
  printf("enter matrix size and distribution block size:\n");
  scanf("%d %d", &size, &nb_distr );
}

MPI_Bcast(&size,     1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&nb_distr, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Create template and objects

When using PLAPACK, one doesn't directly specify how matrices and vectors are distributed. Rather, one creates a "template" that discribes matrix and vector distribution. When creating matrices and vectors, one subsequently indicates alignment to this template. A typical user who just wants to solve a problem need not concern her/himself with this. The first parameter passed to `PLA_Temp_create` indicates the distribution block size to be used (see Section 2.1). The second paramater indicates whether indixing will start with 0 (as is customary in C) or 1 (as is customary in FORTRAN). A datastructure that encodes the distribution is returned in the last parameter.

```
PLA_Temp_create( nb_distr, 0, &templ );
```

Since we intend to solve the problem $Ax = b$, we must next create "linear algebra objects" in which to store the data related to A, x, and b. An matrix object to store `size` $\times$ `size` matrix A is created by indicating the datatype, the row and column dimensions, the template for distribution, and the row and column alignments to the template. A datastructure describing the distributed matrix is returned in the last parameter. Vectors x and b are created similarly.

```
PLA_Matrix_create(  MPI_DOUBLE, size, size,
                    templ, PLA_ALIGN_FIRST, PLA_ALIGN_FIRST, &A );

PLA_Mvector_create( MPI_DOUBLE, size, 1, templ, PLA_ALIGN_FIRST, &x );

PLA_Mvector_create( MPI_DOUBLE, size, 1, templ, PLA_ALIGN_FIRST, &b );
```

Note: *Vectors x and b are created as "multivectors" (panels of equally sized vectors) rather than vectors. In a recent rewrite of PLAPACK, it was decided that vectors are special cases of multivectors, and much of the infrastructure was streamlined. Unfortunately, on rare occasion, we now run into problems with vectors, and thus it is much safer to create vectors as multivectors of width one.*

Create a few constants we will need

Frequently when call PLAPACK "Global BLAS" (parallel BLAS operations), we need to pass in scaling constants, which are often $-1, 0$, or 1. The following routine creates multiscalar objects duplicated on all nodes with those values.

```
PLA_Create_constants_conf_to( A, &minus_one, NULL, NULL );
```

Create a problem to solve

We conveniently hide the filling of objects A, x, and b in a subroutine, an example of which is described in Section 2.2

```
create_problem( A, x, b );
```

Factor the matrix

In solving $Ax = b$, we will first compute the Cholesky factorization $A \rightarrow LL^T$ where L is lower triangular. The routine `PLA_Chol` overwrites the lower triangular portion of A with L. Since one is often interested in performance of parallel algorithms, we start timing before the factorization.

```
MPI_Barrier( MPI_COMM_WORLD );
time = MPI_Wtime( );

info = PLA_Chol( PLA_LOWER_TRIANGULAR, A );
```

Solve the problem

If the Cholesky factorization completes successfully (`info` $\neq 0$), $Ax = b$ is solved by observing that $LL^T x = b$ so that one can first solve $Ly = b$ followed by $L^T x = y$. In the following section of code, we call the PLAPACK Global BLAS routine for solving a triangular system of equations `PLA_Trsv`. It is the Cholesky factorization together with the triangular solves that are of interest, and thus we stop timing after these solves.

```
    if ( info != 0 ) {
      printf("nonpositive definite matrix detected\n");
    }
    else {
      PLA_Trsv( PLA_LOW_TRIAN, PLA_NO_TRANS, PLA_NONUNIT_DIAG, A, b );
      PLA_Trsv( PLA_LOW_TRIAN, PLA_TRANS,    PLA_NONUNIT_DIAG, A, b );

      MPI_Barrier( MPI_COMM_WORLD );
      time = MPI_Wtime() - time;
```

Report performance

Computation of the Cholesky factorization of an $n \times n$ dense matrix requires approximately

$$\frac{1}{3}n^3$$

operations. Thus, the number of MFLOPS (Millions of floating point operations) performed per second by our algorithm is given by

$$\frac{1}{3}\texttt{size}^3/\texttt{time} * 10^{-6}$$

Finally, the performance per processor, in MFLOPS/sec per processor, is given by the formula

$$\frac{1}{3}\texttt{size}^3/\texttt{time} * 10^{-6}/\texttt{nprocs}$$

```
      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
      if ( me == 0 ) {
        MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
        printf("n = %d, time = %f, MFLOPS/node = %f\n", size, time,
               1.0/3.0 * size * size * size / time * 1.0e-6 / nprocs );
      }
```

Report the answer

It is a curious fact that often people compute an answer because they are actually interested in it. (Some of us just care about performance.) We hide the extraction of the answer in subroutine `process_answer` discussed in detail in Section 2.3.

```
      process_answer( b );
```

Check accuracy

Once b has been overwritten by a computed approximation to x, we can check how accurate our answer is by computing $||b - x||_2$.

```
      PLA_Axpy( minus_one, x, b );

      PLA_Mscalar_create( MPI_DOUBLE, PLA_ALL_ROWS, PLA_ALL_COLS,
                          1, 1, templ, &b_norm );

      PLA_Nrm2( b, b_norm );
```

```
    if ( me == 0 ) {
      PLA_Obj_get_local_contents( b_norm, PLA_NO_TRANS, &dummy, &dummy,
                                  &b_norm_value, 1, 1 );
      printf( "Norm of difference between original x and computed x: %e \n",
               b_norm_value );
    }
  }
```

Clean up the environment

When creating linear algebra objects, the underlying infrastructure creates space to store the information. The following calls release that space. Similarly, the template is released. Finally, PLAPACK and MPI are finalized.

```
  /* Free to linear algebra objects */
  PLA_Obj_free(&A);              PLA_Obj_free(&x);
  PLA_Obj_free(&b);              PLA_Obj_free(&minus_one);
  PLA_Obj_free(&b_norm);

  /* Free the template */
  PLA_Temp_free(&templ);

  /* Finalize PLAPACK and MPI */
  PLA_Finalize( );
  MPI_Finalize( );
}
```

2.2 Building a problem

The sample program given in Fig. 2.3 shows how to build a linear system to be solved by the previously discussed driver. Again, we step through this sample code:

Subroutine to build problem

The driver routine discussed above passes three linear algebra objects to this routine: `A`, `x`, and `b`. The first is assumed to be a matrix object, while the others are multivectors of width one. The subroutine creates A, x, and b such that $Ax = b$.

```
void create_problem( PLA_Obj A, PLA_Obj x, PLA_Obj b )
{
```

Some useful information

To generate the problem, we need the dimension of matrix A, which equals the lengths of vectors x and b. Also, each processor will compute part of the total matrix, and thus we must extract the node index and the total number of nodes involved in the computation.

```
  PLA_Obj_global_length( A, &size );              /* Extract dimension of A */

  MPI_Comm_rank( MPI_COMM_WORLD, &me );
  MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
```

```
#include "mpi.h"
#include "PLA.h"

void create_problem( PLA_Obj A, PLA_Obj x, PLA_Obj b )
{
  PLA_Obj  zero = NULL,    one = NULL;
  int size, me, nprocs, i, j;
  double *locA, d_one = 1.0;

  PLA_Obj_global_length( A, &size );

  MPI_Comm_rank( MPI_COMM_WORLD, &me );
  MPI_Comm_size( MPI_COMM_WORLD, &nprocs );

  PLA_Create_constants_conf_to( A, NULL, &zero, &one );

  PLA_Obj_set_to_zero ( A );        PLA_Obj_set_to_zero ( x );
  PLA_Obj_set_to_zero ( b );

  PLA_API_begin();
    PLA_Obj_API_open(A);            PLA_Obj_API_open(x);

    srand48( me * 1793 );

    locA = (double *) malloc( sizeof( double ) * size  );
    for (j=me;j< size; j+=nprocs) {
      for (i=0; i < size; i++) locA[i]=drand48() * 2.0 - 1.0;
      locA[j] += size;
      PLA_API_axpy_matrix_to_global( size, 1, &d_one, locA, size, A, 0, j );
    }

    if (0==me) {
      for (i=0; i<size; i++) locA[i]=drand48() * 2.0 - 1.0;
      PLA_API_axpy_vector_to_global( size, &d_one, locA, 1, x, 0);
    }
    free( locA );

    PLA_Obj_API_close(x);           PLA_Obj_API_close(A);
  PLA_API_end();

  PLA_Symv( PLA_LOW_TRIAN, one, A, x, zero, b );

  PLA_Obj_free( &zero );         PLA_Obj_free( &one );
}
```

Figure 2.3: A sample subroutine for creating a symmetric positive definite matrix A and vectors x and b s.t. $Ax = b$.

Initializing A and x

The PLAPACK Application Interface (API) builds a matrix or vector by adding to the existing values in the objects. Thus, we start by initializing A and x to equal zero.

```
PLA_Obj_set_to_zero ( A );         PLA_Obj_set_to_zero ( x );
PLA_Obj_set_to_zero ( b );
```

Application Interface activation

The PLAPACK Application Interface uses one-sided communication to perform the communication necessary to scatter data to global objects. Since this interface only requires a standard compliant MPI(-1) implementation, under the covers the API uses a mechanism that transparently sends messages and polls for their arrival. The current PLAPACK implementation uses the Managed Message-Passing Interface (MMPI) developed by Carter Edwards at the University of Texas at Austin [**?**]. To start this mechanism, we initiate the API with a call to `PLA_API_begin`.

```
PLA_API_begin();
```

Upon completion, we leave the API mode by calling `PLA_API_end()`.

```
PLA_API_end();                          /* leave Application Interface mode */
```

Caution: *There are limitations on the kinds of operations that can be performed while in API active mode. See* Using PLAPACK *[**?**].*

Opening and closing objects

Before accessing a linear algebra object using the API, it must be opened using a call to `PLA_Obj_API_open( )`.

```
PLA_Obj_API_open(A);              PLA_Obj_API_open(x);
```

Once all contributions to a linear algebra object have been submitted (as discussed below), the object is close by a call to `PLA_Obj_API_close()`.

```
PLA_Obj_API_close(x);              PLA_Obj_API_close(A);
```

Caution: *Contributions to a linear algebra object can NOT assumed to actually have been written until a call to* `PLA_Obj_API_close()` *or* `PLA_Obj_API_sync` *has been made.*

Submitting data to the objects

We now give an example that builds a random matrix. Let A be $n \times n$ and partition A by columns:

$$A = \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{n-1} \end{array} \right)$$

The following code generates the columns on the different nodes in a round-robin fashion. In other words, column a_i is generated by node $i \bmod p$ where p equals the number of processors in the mesh. Once computed, a column is submitted to the global matrix by a call to `PLA_API_axpy_matrix_to_global`, with calling sequence

```
PLA_API_axpy_matrix_to_global( int m, int n,
                               void *alpha, void *A_local, int lda_local,
                               PLA_Obj A_local, int displ_i, int displ_j )
```

The operation performed (using MATLAB-like syntax) is

$$A_{\text{global}}(i : i+m-1, j : j+n-1) = \alpha A_{\text{local}} + A_{\text{global}}(i : i+m-1, j : j+n-1)$$

Thus,

- `m` and `n` are the row and column dimensions of the matrix computed by this node that is to be *added* to the global object.
- `alpha` gives the address where the multiplier α can be found. The datatype of this scalar is determined by the datatype of `A_global`.
- The matrix to be added can be found in array at address `A_local`, with a leading dimension `lda_local`, stored column-major (FORTRAN-like). The data is to be added to the global matrix at upper-left-element `( displ_i, displ_j )`. Notice: whether indexing starts at 0 or 1 depends on how the template was created for object `A_global`.

```
    srand48( me * 1793 );

    locA = (double *) malloc( sizeof( double ) * size  );
    for (j=me;j< size; j+=nprocs) {
      for (i=0; i < size; i++) locA[i]=drand48() * 2.0 - 1.0;
      locA[j] += size;
      PLA_API_axpy_matrix_to_global( size, 1, &d_one, locA, size, A, 0, j );
    }
```

Notice: *a call to* `PLA_API_axpy_matrix_to_global`

is a one-sided call. It submits a contribution to the global matrix. The data is passed to the global matrix asynchronously, and cannot be assumed to have been added until a call to `PLA_Obj_API_close()` *or* `PLA_Obj_API_sync` *has been made.*

Notice: *to ensure that the matrix is positive definite, a large quantity (= the matrix dimension) is added to the diagonal of the matrix.*

In our example, node 0 computes a solution vector, reusing the allocated space `locA` and submits it similarly to a global vector `x` .

```
    if (0==me) {
      for (i=0; i<size; i++) locA[i]=drand48() * 2.0 - 1.0;
      PLA_API_axpy_vector_to_global( size, &d_one, locA, 1, x, 0);
    }
    free( locA );
```

Computing $b = Ax$

```
  PLA_Symv( PLA_LOW_TRIAN, one, A, x, zero, b );
```

Clean up the environment

```
  PLA_Obj_free( &zero );          PLA_Obj_free( &one );
}
```

2.3 Getting the Answer

In the previous section, we showed how the PLAPACK Application Interface can be used to fill a matrix or vector using one-sided communication. In this section, we show how it can be used similarly to extract information from global matrices or vectors. The example in Fig. 2.4 shows how node 0 extracts the contents of global vector `x` to a local buffer after which it is "processed". In this example, this means the first and last entry are printed.

2.4 Putting it all together

In this section, we show how to intall PLAPACK, the solver library, and the tutorial. While we use the Cray T3E as an example, installation on other systems should be similar to this case.

Installing PLAPACK

- Obtain the latest PLAPACK release via anonymous ftp:

```
% ftp ftp.cs.utexas.edu
Name (ftp.cs.utexas.edu:u20009): anonymous
Password:
ftp> cd pub/plapack/R?.?
ftp> ls
ftp> bin
ftp> get plapackR??.t3e.tar.Z
ftp> quit
221 Goodbye.
```

- Uncompress and un-"tar":

```
% uncompress plapackR??.t3e.tar.Z
% tar -xf plapackR??.t3e.tar
```

- This creates a new directory `PLAPACK`:

```
% cd PLAPACK
% ls
```

Modifying the Makefile

Makefiles can be highly platform specific. To keep one from having to worry about these specifics, all platform specific definitions have been consolidated into the file `Make.include`, which is included in subsequent makefiles. Since a lot of the definitions are relative to the PLAPACK root directory, `Make.include` must be modified so that the first line identifies this root directory.

- Find out the path to the PLAPACK root directory:

```
~/PLAPACK & pwd
/home/utexas/cs/u20009/PLAPACK
```

- Using your favorite editor, enter `Make.include` and modify the first lines:

```
# Top level PLAPACK directory.
PLAPACK_ROOT = /home/utexas/cs/u20009/PLAPACK
```

```c
#include "PLA.h"

void process_answer( PLA_Obj x )
{
  int size, me, i;
  double *locx, d_one = 1.0;

  MPI_Comm_rank( MPI_COMM_WORLD, &me );

  PLA_API_begin();
    PLA_Obj_API_open( x );

    if ( me == 0 ) {
      PLA_Obj_global_length( x, &size );
      locx = (double *) malloc( sizeof( double ) * size  );
      for (i=0; i<size; i++) locx[i] = 0.0;

      PLA_API_axpy_global_to_vector( size, &d_one, x, 0, locx, 1);
    }

    PLA_Obj_API_sync( x );

    if ( me == 0 ) {
      printf("first computed entry: %f\n", locx[0] );
      printf("last  computed entry: %f\n", locx[size-1] );

      free( locx );
    }

    PLA_Obj_API_close(x);
  PLA_API_end();
}
```

Figure 2.4: A sample subroutine for processing the answer of the solve.

Installing the solver library

PLAPACK is billed as an infrastructure for building parallel linear algebra libraries. Thus, while we have created a few solvers, we consider these to be separate from PLAPACK itself. Thus, one must obtain the solver package separately:

- From the PLAPACK root directory, obtain the solver package, uncompress, and un-"tar":

  ```
  ~/PLAPACK % ftp ftp.cs.utexas.edu
  Name (ftp.cs.utexas.edu:u20009): anonymous
  Password:
  ftp> cd pub/plapack/SOLVERS/R?.?
  ftp> bin
  ftp> get solversR??.tar.Z
  ftp> quit
  ~/PLAPACK % uncompress solversR??.tar.Z
  ~/PLAPACK % tar -xf solversR??.tar
  ```

  This creates a subdirectory `SOLVERS`.

- Within the solver subdirectory, build the solver library:

  ```
  ~/PLAPACK % cd SOLVERS
  ~/PLAPACK/SOLVERS % make
  ```

  Notice: the makefiles are set up so that all should work *as long as SOLVERS is a subdirectory of the PLAPACK root directory*.

- The above operations automatically create a library `libPLAPACK_SOLVERS.a` in the PLAPACK root directory.

Examples

The examples given in this chapter can be installed as follows:

```
~/PLAPACK % ftp ftp.cs.utexas.edu
Name (ftp.cs.utexas.edu:u20009): anonymous
Password:
ftp> cd pub/plapack/EXAMPLES/
ftp> bin
ftp> get Tutorial.tar.Z
ftp> quit
~/PLAPACK % uncompress Tutorial.tar.Z
~/PLAPACK % tar -xf Tutorial.tar.Z
```

This will create a directory `Tutorial`. Next, perform the following commands

```
~/PLAPACK % cd Tutorial
~/PLAPACK/Tutorial % cd chol/C/Chapter2
~/PLAPACK/Tutorial/chol/C/Chapter2 % make
~/PLAPACK/Tutorial/chol/C/Chapter2 % mpprun -n 4 Chol_driver.x
4000 24
n = 4000, time = 22.275862, MFLOPS/node = 239.422091
first computed entry: -0.922733
last  computed entry: -0.240827
Norm of difference between original x and computed x: 2.305625e-14
```

Notice that `mpprun` is the Cray specific command for executing an MPI program. The `-n 4` option indicates four nodes are to be created. The problem size created by the driver is 200×200, which is distributed using a template blocking size of 24.

Chapter 3

Zen and the Art of Optimizing the Sequential Cholesky Factorization

The Cholesky factorization of an $n \times n$ matrix A is given by

$$A \rightarrow LL^T$$

where A is symmetric positive definite and L is lower triangular. We will discuss implementation of two approaches to computing this factorization, sometimes known as the right-looking and left-looking variants.

3.1 Algorithms

3.1.1 Right-looking variant

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

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

where α_{11} and λ_{11} are scalars. As a result, a_{21} and l_{21} are vectors of length $n-1$, and A_{22} and L_{22} are matrices of size $(n-1) \times (n-1)$. The $\star$ indicates the symmetric part of A, which will not be updated. Now,

$$A = \left(\begin{array}{c|c} a_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} \lambda_{11} & l_{21}^T \\ \hline 0 & L_{22}^T \end{array}\right) = \left(\begin{array}{c|c} \lambda_{11}^2 & \star \\ \hline \lambda_{11} l_{21} & l_{21} l_{21}^T + L_{22} L_{22}^T \end{array}\right)$$

From this we derive the equations

$$\begin{array}{rcl} \lambda_{11} & = & \sqrt{\alpha_{11}} \\ l_{21} & = & a_{21}/\lambda_{11} \\ A_{22} - l_{21} l_{21}^T & = & L_{22} L_{22}^T \end{array}$$

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. $\alpha_{11} \leftarrow \lambda_{11} = \sqrt{\alpha_{11}}$
3. $a_{21} \leftarrow l_{21} = a_{21}/\lambda_{11}$
4. $A_{22} \leftarrow A_{22} - l_{21} l_{21}^T$
5. continue recursively with A_{22}

Note 3.1.1 *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-1 update.

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

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

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

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

so that

$$\begin{array}{rcl} A_{TL} & = & L_{TL} L_{TL}^T \\ A_{BL} & = & L_{BL} L_{TL}^T \\ A_{BR} & = & L_{BR} L_{BR}^T + L_{BL} L_{BL}^T \end{array}$$

It can be easily verified that the above algorithm has the effect that after k steps

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

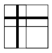
Thus, the matrix with which the algorithm is continued at each step is the submatrix $A_{BR}$and to complete the Cholesky factorization, it suffices to compute the factorization of the updated A_{BR}. This motivates the algorithm given in Fig. 3.1.

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

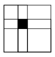
where A_{TL} is 0×0

do until A_{BR} is 0×0

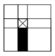
repartition



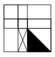
$$\left(\begin{array}{c||c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) = \left(\begin{array}{c||c|c} A_{00} & \star & \star \\ \hline\hline a_{10}^T & \alpha_{11} & \star \\ \hline A_{20} & a_{21} & A_{22} \end{array} \right)$$



$\alpha_{11} \leftarrow \lambda_{11} = \sqrt{\alpha_{11}}$

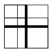


$a_{21} \leftarrow l_{21} = a_{21}/\lambda_{11}$



$A_{22} \leftarrow A_{22} - l_{21} l_{21}^T$

continue with



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

enddo

Figure 3.1: Right-looking variant of the Cholesky factorization algorithm.

3.2 Sequential Implementation: No Frills

3.2.1 Right-looking Variant

Implementation

A naieve approach to implementing the described right-looking algorithm is to implement the individual operations as loops of the data. Thus, in the algorithm in Fig. 3.1, the outer-most loop keeps track of the subproblem yet to be factored. Inside of this loop, there is a second loop that implements the scaling of $a_{21} \leftarrow a_{21}/\alpha_{11}$ and a double loop that implements the symmetric rank-1 update $A_{22} \leftarrow A_{22} - a_{21}a_{21}^T$. This is illustrated in Fig. 3.2, with the algorithm already given in Fig. 3.1 and the algorithm that exposes the loops explicitely presented on the left and right, respectively. In that figure, we also present simple FORTRAN and C codes.

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

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

$\alpha_{11} \leftarrow \lambda_{11} = \sqrt{\alpha_{11}}$

$a_{21} \leftarrow l_{21} = a_{21} / \lambda_{11}$

$A_{22} \leftarrow A_{22} - l_{21} l_{21}^T$

continue with

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

enddo

(a) Original right-looking algorithm

for $j = 1, n$
  $\alpha_{j,j} = \sqrt{\alpha_{j,j}}$
  for $i = j+1, n$
    $\alpha_{i,j} = \alpha_{i,j} / \alpha_{j,j}$
  endfor
  for $k = j+1, n$
    for $i = k, n$
      $\alpha_{i,k} = \alpha_{i,k} - \alpha_{i,j} \alpha_{k,j}$
    endfor
  endfor
enddo

(b) Reformulation using triply nested loops.

```
SUBROUTINE CHOL_RIGHT_0( N, A, LDA )
INTEGER N, LDA
DOUBLE PRECISION A( LDA, * )

DO J=1,N
  A(J,J) = SQRT(A(J,J))
  DO I=J+1,N
    A(I,J) = A(I,J)/A(J,J)
  ENDDO
  DO K=J+1,N
    DO I=K,N
      A(I,K) = A(I,K)-A(I,J)*A(K,J)
    ENDDO
  ENDDO
ENDDO

RETURN
END
```

(c) FORTRAN implementation

```
#include <math.h>

#define A(i,j) a[ ((j)-1)*(lda)+(i)-1 ]

void chol_right_0
             (int n, double *a, int lda)
{
  int i, j;

  for (j=1; j<=n; j++){
    A(j,j) = dsqrt( A(j,j) );
    for (i=j+1;i<=n;i++)
      A(i,j) = A(i,j)/A(j,j);
    for (k=j+1;j<=n;j++)
      for (i=k;i<=n;i++)
         A(i,k) = A(i,k)-A(i,j)*A(k,j);
  }
  return;
}
```

(d) C implementation

Figure 3.2: Right-looking Cholesky factorization implemented using triply nested loops.

3.3 Sequential Implementation: Level 1 BLAS

3.3.1 Right-looking Variant

How the algorithm given above is implemented can greatly affect the performance of the code. To attain portability and some level of performance, LINPACK implements such algorithms using *vector-vector* operations. These vector-vector operations are commonly used and highly optimized versions are provided by most vendors as part of the (Level-1) Basic Linear Algebra Subprograms [**?**].

3.3.1.1 Implementation

Clearly, the square-root operation performed is merely an operation on a scalar. The scaling of the rest of the first column, $a_{21} \leftarrow a_{21}/\lambda_{11}$ is an operation on a vector. For this, the (Level-1) Basic Linear Algebra Subprograms provide the routine □SCAL, where chosing □ equal to {S,D,C,Z} allows one to specify Single, Double, Complex or double complex (Z) arithmetic. Finally, there are a number of ways of implementing the symmetric rank-1 update $A_{22} \leftarrow A_{22} - l_{21} l_{21}^T$: To derive a vector-vector implementation, one can partition A_{22} by columns and l_{21} elementwise:

$$A_{22} = \left(\begin{array}{c|c|c|c} a_1^{(22)} & a_2^{(22)} & \cdots & a_{n-1}^{(22}) \end{array} \right) \quad \text{and} \quad l_{21} = \left(\begin{array}{c} \lambda_1^{(21)} \\ \hline \lambda_2^{(21)} \\ \hline \vdots \\ \hline \lambda_{n-1}^{(21)} \end{array} \right)$$

Then the symmetric rank-1 update becomes

$$\begin{aligned} A_{22} - l_{21} l_{21}^T &= \left(\begin{array}{c|c|c|c} a_1^{(22)} & a_2^{(22)} & \cdots & a_{n-1}^{(22}) \end{array} \right) - l_{21} \left(\begin{array}{c|c|c|c} \lambda_1^{(21)} & \lambda_2^{(21)} & \cdots & \lambda_{n-1}^{(21)} \end{array} \right) \\ &= \left(\begin{array}{c|c|c|c} a_1^{(22)} - \lambda_1^{(21)} l_{21} & a_2^{(22)} - \lambda_2^{(21)} l_{21} & \cdots & a_{n-1}^{(22} - \lambda_{n-1}^{(21)} l_{21}) \end{array} \right) \end{aligned}$$

In other words, each column of A_{22} is updated by subtracting a multiple of l_{21}. This operation is commonly called an □AXPY operation, for scalar Alpha times vector X Plus vector Y. The double precision version of this operation (DAXPY) is a frequently used computational kernel provided as part of the Level-1 BLAS. Notice that if only the lower triangular portion of A_{22} is to be updated, only part of each column is updated. In Fig. 3.4 (a), we give the original right-looking algorithm. In (b) of that figure, we give a "MATLAB"-like algorithm that replaces the loop that scales a_{21} and the innermost loop of the symmetric rank-1 update by vector-vector operations. FORTRAN and C implementations are given in (c) and (d).

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

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

$\alpha_{11} \leftarrow \lambda_{11} = \sqrt{\alpha_{11}}$

$a_{21} \leftarrow l_{21} = a_{21} / \lambda_{11}$

$A_{22} \leftarrow A_{22} - l_{21} l_{21}^T$

continue with

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

enddo

(a) Original right-looking algorithm

for $j = 1, n$

$\alpha_{j,j} = \sqrt{\alpha_{j,j}}$

$\alpha_{j+1:n,j} = \alpha_{j+1:n,j} / \alpha_{j,j}$

for $k = j+1, n$

$\alpha_{k:n,k} = \alpha_{k:n,k} - \alpha_{k,j} \alpha_{k:n,j}$

endfor

enddo

(b) Algorithm reformulated to use vector operations in the inner-most loops

```
      SUBROUTINE CHOL_RIGHT_1( N, A, LDA )
      INTEGER N, LDA
      DOUBLE PRECISION A( LDA, * )

      DO J=1,N
        A(J,J)=SQRT(A(J,J))
        CALL DSCAL(N-J,1.0D00/A(J,J),A(J+1,J),1)
        DO K=J+1,N
          CALL DAXPY(N-J+1,-A(K,J),A(K,J),1,
     $                            A(K,K),1)
        ENDDO
      ENDDO

      RETURN
      END
```

(c) FORTRAN implementation

```
#include <math.h>

#define A(i,j) a[ ((j)-1)*(lda)+(i)-1 ]

void chol_right_1
             (int n, double *a, int lda)
{
  int i, j, k, nmj, nmkp1, i1=1;
  double Ajj_inv, min_Akj;

  for (j=1; j<=n; j++){
    A(j,j) = dsqrt( A(j,j) );
    Ajj_inv = 1.0d00/A(j,j);
    dscal(&nmj,&Ajj_inv,&A(j+1,j),&i1);
    for (k=j+1;j<=n;j++){
      nmkp1 = n-k+1;
      min_Akj = -A(k,j);
      daxpy(&nmkp1,&min_Akj,&A(k,j),&i1,
                              &A(k,k),&i1 );
    }
  }
  return;
}
```

(d) C implementation

Figure 3.3: Right-looking Cholesky factorization implemented using doubly nested loops and vector-vector operations.

3.4 Sequential Implementation: Level 2 BLAS

3.4.1 Right-looking Variant

3.4.1.1 Implementation

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

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

$\alpha_{11} \leftarrow \lambda_{11} = \sqrt{\alpha_{11}}$

$a_{21} \leftarrow l_{21} = a_{21}/\lambda_{11}$

$A_{22} \leftarrow A_{22} - l_{21} l_{21}^T$

continue with

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

enddo

(a) Original right-looking algorithm

for $j = 1, n$

$\alpha_{j,j} = \sqrt{\alpha_{j,j}}$

$\alpha_{j+1:n,j} = \alpha_{j+1:n,j}/\alpha_{j,j}$

$\alpha_{j+1:n,j+1:n} = \alpha_{j+1:n,j+1:n} - \alpha_{j+1:n,j}\alpha_{j+1:n,j}^T$

enddo

(b) Algorithm reformulated to use symmetric rank-1 update. (Notice: only lower triangular part of A is updated.)

```
      SUBROUTINE CHOL_RIGHT_2( N, A, LDA )
      INTEGER N, LDA
      DOUBLE PRECISION A( LDA, * )

      DO J=1,N
        A(J,J) = SQRT(A(J,J))
        CALL DSCAL(N-J,1.0D00/A(J,J),A(J+1,J),1)
        CALL DSYR("Lower triang",
     $            N-J,-1.0D00,A(J+1,J),1,
     $                        A(J+1,J+1),LDA)
      ENDDO

      RETURN
      END
```

(c) FORTRAN implementation

```
#include <math.h>

#define A(i,j) a[ ((j)-1)*(lda)+(i)-1 ]

void chol_right_2
             (int n, double *a, int lda)
{
  int i, j, k, nmj, i1=1;
  double Ajj_inv, dneg1=-1.0d00;

  for (j=1; j<=n; j++){
    A(j,j) = dsqrt( A(j,j) );
    Ajj_inv = 1.0d00/A(j,j);
    dscal(&nmj,&Ajj_inv,&A(j+1,j),&i1);
    dsyr("Lower triang",
         &nmj,&dneg1,&A(j+1,j),&i1,
                     &A(j+1,j+1),&lda );
  }
  return;
}
```

(d) C implementation

Figure 3.4: Right-looking Cholesky factorization implemented using matrix-vector operations.

3.5 Sequential Implementation: Level 3 BLAS

3.5.1 Right-looking Variant

3.5.1.1 Implementation

While the Level-1 and Level-2 BLAS operations represent a useful set of kernels and vendors generally provide highly optimized implementations, there is an inherent limitation in the performance they can attain. Vector-vector operations provided by the Level-1 BLAS perform $O(n)$ operations on $O(n)$ data, where n equals the length of the vectors involved. As a result, they require the same order of magnitude memory operations as floating point operations. Since in current microprocessors memory operations are typically considerably slower than floating point operations, they become an inherent bottleneck. Adding a cache to the memory hierarchy does not generally provide any relieve, since there tends to be no data reuse in these operations.

Although there is an opportunity for data reuse in the matrix-vector operations provided by the Level-2 BLAS, these operations perform $O(n^2)$ operations on $O(n^2)$ data [1], where n now equals the matrix dimension. Thus memory operations can, and often will, still become a bottleneck.

Observe that when performing a matrix-matrix operation, $O(n^3)$ operations are performed on $O(n^2)$ data, where n equals the dimensions of the matrix. Thus, there is an opportunity to bring (parts of) the matrices into fast cache memory and perform (a part of) the matrix-matrix multiplication before returning the result to main memory. Thus, matrix-matrix operations, given by Level-3 BLAS, can in theory and practice achieve near-peak performance.

Given the derivation of the matrix-vector operation (`syr`) algorithm given in Section **??**, it is not difficult to derive a matrix-matrix based operation, as we show next. We purposely follow the discussion in Section **??**.

The level-3 BLAS 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 now $b \times b$, where $b << n$ (say 32 or 64). 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

$$\begin{array}{rcl} L_{11} & = & \text{Cholesky factor of } A_{11} \\ L_{21} & = & A_{21}L_{11}^{-1} \\ A_{22} - L_{21}L_{21}^T & = & L_{22}L_{22}^T \end{array}$$

A blocked 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}

[1] Notice: multiplying an $n \times n$ matrix times a vector requires approximately $2n^2$ floating point operations and approximately n^2 memory operations

Note 3.5.1 *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*, where k here equals b.*

Partitioning

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

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

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

so that

$$\begin{array}{rcl} A_{TL} & = & L_{TL}L_{TL}^T \\ A_{BL} & = & L_{BL}L_{TL}^T \\ A_{BR} & = & L_{BR}L_{BR}^T + L_{BL}L_{BL}^T \end{array}$$

It can be easily verified that the above algorithm has the effect that if the recursion has proceed so that the factorization left to be done involves only A_{BR},

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

This motivates the formulation of the algorithm given in Fig. 3.6.

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

Assume A_{TL} and A_{BL} have been overwritten by L_{TL} and L_{BL} respectively, and A_{BR} has been overwritten by $A_{BR}^{\text{new}} = A_{BR} - L_{BL}L_{BL}^T$ so that only the Cholesky factorization of A_{BR}^{new} needs to be computed.

$$A_{BR}^{\text{new}} = \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right) \quad \text{and} \quad L_{BR} = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right)$$

where α_{11} and λ_{11} are scalars. Then

$$\begin{aligned} A_{BR}^{\text{new}} &= \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right) \\ &= \left(\begin{array}{c|c} \lambda_{11}^2 & \star \\ \hline \lambda_{11} l_{21} & l_{21} l_{21}^T + L_{22} L_{22}^T \end{array}\right) = L_{BR} L_{BR}^T \end{aligned}$$

From this we derive the equations

$$\begin{aligned} \lambda_{11} &= \sqrt{\alpha_{11}} \\ l_{21} &= a_{21}/\lambda_{11} \\ A_{22} - l_{21} l_{21}^T &= L_{22} L_{22}^T \end{aligned}$$

The above motivation yields the 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

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

$\alpha_{11} \leftarrow \lambda_{11} = \sqrt{\alpha_{11}}$

$a_{21} \leftarrow l_{21} = a_{21}/\lambda_{11}$

$A_{22} \leftarrow A_{22} - l_{21} l_{21}^T$

continue with

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

enddo

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

where A_{11} and L_{11} are of size $b \times b$. Then

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

From this we derive the equations

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

The above motivation yields the 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
determine block size b
repartition

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

where A_{11} 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

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

enddo

Figure 3.5: Similarity in the derivation of a Level-2 (*left*) and Level-3 (*right*) BLAS based right-looking Cholesky factorization.

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

determine block size b

repartition

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

where A_{11} 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

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

enddo

(a) Original right-looking algorithm

for $j = 1, n, nb$

determine block size b

$\alpha_{j:j+b-1,j:j+b-1} \leftarrow \lambda_{j:j+b-1,j:j+b-1}$

$= \text{Chol. fact.}(\alpha_{j:j+b-1,j:j+b-1})$

$\alpha_{j+b:n,j:j+b-1} = \alpha_{j+b:n,j:j+b-1} \lambda_{j:j+b-1,j:j+b-1}^{-T}$

$\alpha_{j+b:n,j+b:n} = \alpha_{j+b:n,j+b:n} -$

$\alpha_{j+b:n,j:j+b-1} \alpha_{j+b:n,j+b-1}^{T}$

enddo

(b) Algorithm reformulated to use matrix-matrix operations

```
 SUBROUTINE CHOL_RIGHT_3( N, NB, A, LDA )
 INTEGER N, LDA, NB, JB
 DOUBLE PRECISION A( LDA, * )

 DO J=1,N,NB
   JB=MIN(NB,N-J+1)
   CALL CHOL_RIGHT_2(JB,A(J,J),LDA)
   IF (J+JB.LE.N) THEN
     CALL DTRSM("Right","Lower triang",
$           "Trans","Nonunit",
$           N-J-JB+1,JB,1.0D00,A(J,J),LDA,
$                            A(J+JB,J),LDA)
     CALL DSYRK("Low. tri-
ang","No trans",
$       N-J-JB+1,JB,-
1.0D00,A(J+JB,J),LDA,
$                 1.0D00,A(J+JB,J+JB),LDA )
   ENDIF
 ENDDO
 RETURN
 END
```

(c) FORTRAN implementation

```
void chol_right_3
     (int n, int nb, double *a, int lda)
{
  int j, jb, nmjmjbp1;
  double d1=1d00, dneg1=-1.0d00;

  for (j=1; j<=n; j+=jb){
    jb=min(nb,n-j+1);
    chol_right_2(jb,&A(j,j),lda);
    if (j+jb<n){
      nmjmjbp1=n-j-jb+1;
      dtrsm("Right","Lower triang",
           "Trans","Nonunit",
           &nmjmjbp1,&jb,&d1,&A(j,j),&lda,
           &A(j+jb,j),&lda);
      dsyrk("Lower triang","No trans",
           &nmjmjbp1,&jb,&dneg1,&A(j+jb,j),
           &lda,&d1,&A(j+jb,j+jb),&lda);
    }
  }
  return;
}
```

(d) C implementation

Figure 3.6: Blocked right-looking Cholesky factorization implemented using level-3 BLAS.

3.6 Performance

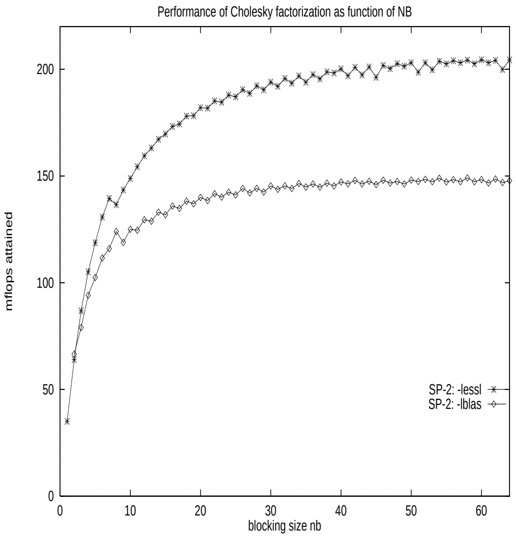


Figure 3.7: Performance as a function of the blocking size nb. The reported performance was attained on one node of an IBM SP-2. Two different BLAS libraries were linked: the "-lblas" library and the more highly optimized "-lessl" library.

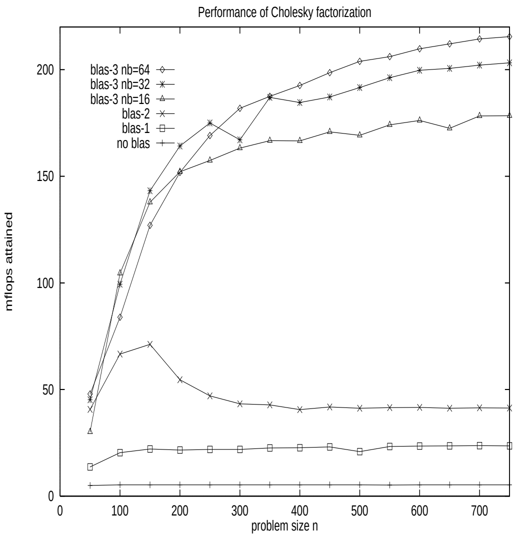


Figure 3.8: Performance of the Cholesky factorization with different levels of optimization. The reported performance was attained on one node of an IBM SP-2, using the -lessl library.

Chapter 4

PLAPACK Cholesky Factorization, Part I

Let us reconsider the algorithm in Fig. 3.1, which we repeated in subsequent figures claiming this to be a natural way of expressing the algorithm. We now also claim that the MATLAB M-script like implementations, as well as the FORTRAN and C implementations, reflect this algorithm rather poorly. Indeed, the abundance of indices make these codes, despite the fact that they are rather short, difficult to read. If it is already difficult to recognize the algorithm in the sequential implementations, one can imagine what would happen if one has to also keep track of the processors on which the matrix is discributed, how the matrix is distributed, etc.

Let us see how to code if we assume that all information about matrices and vectors has been hidden in *opaque objects*, as illustated below:

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

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

$\alpha_{11} \leftarrow \lambda_{11} = \sqrt{\alpha_{11}}$

$a_{21} \leftarrow l_{21} = a_{21} / \lambda_{11}$

$A_{22} \leftarrow A_{22} - l_{21} l_{21}^T$

continue with

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

enddo

Figure 4.1: Right-looking variant of the Cholesky factorization algorithm, and how one may wish to encode it.

The code on the right is almost identical to PLAPACK code for this algorithm. Notice that the algorithm as we claim it is naturally expressed can be easily recognized in the implementation, even if a reader has never seen PLAPACK code before.

In subsequent sections, we will discuss the basic PLAPACK implementations of this algorithm, and how to progressively optimize to attain extremely high performance on a wide range of distributed memory parallel architectures.

4.1 A Parallel Level-2 BLAS Based Implementation

A complete implementation of a PLAPACK level-2 BLAS right-looking variant of the Cholesky factorization is given in Fig. 4.2. The primary part of the code is given in lines 19–34. Notice that this is essentially the same as was given in Fig. 4.1 except that the view of the current diagonal element, `A11`, is cast to be a multiscalar (indeed, it is really a scalar) and a check is made for make sure no square root of a negative number is encountered. The casting of `A11` is required since the first parameter passed to `PLA_Inv_scal` must be a multiscalar. Also, at this time, the `PLA_Syr` routine cannot take a view of a matrix as input for the vector being used for the symmetric rank-1 update. Thus, we use a call to `PLA_Syrk`, the symmetric rank-k update, with one of operands of unit width.

A common parameter passed to PLAPACK global and local BLAS routines are the constants "0", "1", and/or "-1". Since these must be passed as multiscalar objects, a typical routine starts by creating those objects. We have introduced a new utility routine

```
PLA_Create_constants_conf_to( PLA_Obj Obj_in, PLA_Obj *minus_one,
                              PLA_Obj *zero,  PLA_Obj *one )
```

which creates the appropriate objects using the same datatype and template as were used to create input object `Obj_in`. Passing in a `NULL` for an output parameter indicates that object need not be created. The remainder of the code is concerned with declarations and the freeing of temporary views.

Note 4.1.1 *This code has a clear shortcoming in that only the node that owns the current diagonal element α_{11} will determine that a square-root of a negative number is to be taken, and thus only that node will recognize an error condition. Rather than fixing this problem now, we will show how to fix it in the ultimate implementation.*

Note 4.1.2 *The performance of this implementation on a 4×4 configuration of the Cray T3E is given in Fig. 4.5. Notice that the maximum performance is less than 2 MFLOPS/sec per node on a machine with a peak performance in the 450 MFLOPS/sec per node range... Thus, there is considerable room for improvement!*

```
#include "PLA.h"
#include "mpi.h"

Chol_right_level2( PLA_Obj A, int *info )
{
  int       size;
  PLA_Obj   ABR        = NULL,            A11         = NULL,
                                          A21         = NULL,
            one        = NULL,            minus_one   = NULL;

  /* Create usual duplicated scalar constants with same datatype and
     template as A */
  PLA_Create_constants_conf_to( A, &minus_one, NULL, &one );

  /* Initially the active part of A, ABR, is all of A */
  PLA_Obj_view_all( A, &ABR );

  while ( TRUE ) {
    /* Determine size of current panel */
    PLA_Obj_global_length( ABR , &size );
    if ( 0 == size ) break;

    PLA_Obj_split_4( ABR, 1, 1,    &A11, PLA_DUMMY,
                                   &A21, &ABR );

    PLA_Obj_objtype_cast( A11, PLA_MSCALAR );

    *info = PLA_Sqrt( A11 );
    if ( *info != 0 ) break;

    PLA_Inv_scal( A11, A21 );
    /* Note: this should be a call to Syr !!! */
    PLA_Syrk( PLA_LOW_TRIAN, PLA_NO_TRANS, minus_one, A21, one, ABR );
  }

  PLA_Obj_free( &ABR );         PLA_Obj_free( &A11 );
  PLA_Obj_free( &A21 );         PLA_Obj_free( &one );
  PLA_Obj_free( &minus_one );
}
```

Figure 4.2: Basic level-2 BLAS based right-looking Cholesky factorization.

```
#include "mpi.h"
#include "PLA.h"

int PLA_Sqrt( PLA_Obj A )
{
  int length, width, info = 0;
  MPI_Datatype datatype;

  PLA_Obj_local_length( A, &length );
  PLA_Obj_local_width ( A, &width );

  if ( length == 1 && width == 1 ) {
    PLA_Obj_datatype( A, &datatype );
    if ( datatype == MPI_DOUBLE ) {
      double *buf;

      PLA_Obj_local_buffer( A, (void **) &buf );
      if ( *buf < 0.0 ) info = -1;
      else *buf = sqrt( *buf );
    }
    else if ( datatype == MPI_FLOAT ) {
      float *buf;

      PLA_Obj_local_buffer( A, (void **) &buf );
      if ( *buf < 0.0 ) info = -1;
      else *buf = sqrt( (double) *buf );
    }
    else info = -1;
  }

  return info;
}
```

Figure 4.3: Utility routine to take square root.

4.2 A Parallel Level-3 BLAS Based Implementation

In Fig. 4.4 we give a complete implementation of a basic PLAPACK level-3 BLAS right-looking Cholesky factorization. Notice that lines 17–31 are essentially identical to those given in Fig. 4.2 except that again a check is added for detecting an encounter of a square-root of a negative number. This routine calls the previously discussed level-2 implementation.

```
1    #include "PLA.h"
2    #include "mpi.h"
3
4    Chol_right_1( PLA_Obj A, int nb_alg, int *info )
5    {
6      int       size;
7      PLA_Obj   ABR             = NULL,     A11             = NULL,
8                A21             = NULL,
9                one             = NULL,     minus_one       = NULL;
10
11     /* Create usual duplicated scalar constants */
12     PLA_Create_constants_conf_to( A, &minus_one, NULL, &one );
13
14     /* View ABR = A */
15     PLA_Obj_view_all( A, &ABR );
16
17     while ( TRUE ) {
18       /* Determine size of current panel */
19       PLA_Obj_global_length( ABR , &size );
20       if ( 0 == ( size = min( size, nb_alg ) ) ) break;
21
22       PLA_Obj_split_4( ABR, size, size,    &A11, PLA_DUMMY,
23                                            &A21, &ABR );
24
25       Chol_right_level2( A11, info );
26       if ( *info != 0 ) break;
27
28       PLA_Trsm( PLA_SIDE_RIGHT, PLA_LOW_TRIAN, PLA_TRANS, PLA_NONUNIT_DIAG,
29                 one, A11, A21 );
30       PLA_Syrk( PLA_LOW_TRIAN, PLA_NO_TRANS, minus_one, A21, one, ABR );
31     }
32
33     PLA_Obj_free( &ABR );        PLA_Obj_free( &A11 );
34     PLA_Obj_free( &A21 );
35     PLA_Obj_free( &one );        PLA_Obj_free( &minus_one );
36   }
```

Figure 4.4: Basic level-3 BLAS based right-looking Cholesky factorization.

Note 4.2.1 *This code continues to have a clear shortcoming in that only the node that owns the current diagonal element* α_{11} *will determine that a square-root of a negative number is to be taken, and thus only that node will recognize an error condition.*

Note 4.2.2 *Fig. 4.5 shows that performance is improved considerably when compared against the level 2 implementation. This is primarily due to the fact that the bulk of the computation is now performed using level-3 BLAS.*

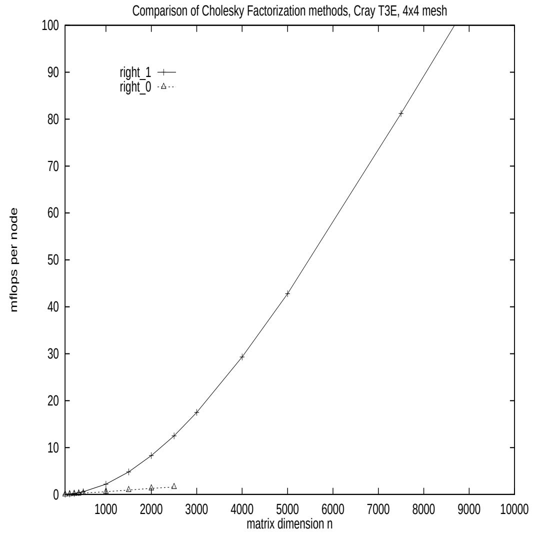


Figure 4.5: Performance of the basic level-2 and level-3 based PLAPACK implementations on a 16 processor configuration of the Cray T3E. A distribution block size of 64 and algorithmic block size of 128 was used to collect this data.

Note 4.2.3 *Looking ahead to Fig. 4.8, we see that performance is still not all that impressive: the code only attains around 100 MFLOPS/sec per node even for very large problem sizes on a machine with a peak performance in the 450 MFLOPS/sec per node range.*

4.3 Factoring A_{11} Local on One Node

As noted in Note 4.2.3, performance of the level-3 based implementation is not particularly impressive, although much better than the level-2 based implemention. The good news is that at least it appears that as the problem size increasing, better performance can be expected.

The primary reason why the level-3 based algorithm does not reach high performance quickly is that it calls the level-2 based parallel algorithm, which inherently communicates a very large number of messages. In addition, many communications performed as part of this smaller factorization are repeated in the global triangular solve with multiple right-hand-side operation performed by `PLA_Trsm`. In turn, many communications performed by `PLA_Trsm` are repeated for the symmetric rank-k update `PLA_Syrk`.

One observation is that if the distribution block size is reasonable and the size of the current diagonal matrix A_{11} is small enough, the matrix to be factored by the level-2 BLAS based implementation is owned entirely by one node, and thus a sequential algorithm can be called on that node to factor that subproblem. This results in no communication being required during the factorization of A_{11}.

To facilitate this, one must first determine how large A_{11} can be while guaranteeing that it exists on one node. PLAPACK provides the routine `PLA_Obj_split_size`. Turning to the code in Fig. 4.6 we see that the determination of the current block size previously accomplished by

```
        PLA_Obj_global_length( ABR , &size );
        if ( 0 == ( size = min( size, nb_alg ) ) ) break;
```

in Fig. 4.4 is replaced by

```
        PLA_Obj_split_size( ABR, PLA_SIDE_LEFT, &size_left, &dummy );
        PLA_Obj_split_size( ABR, PLA_SIDE_TOP,  &size_top,  &dummy );
        if ( 0 == ( size = min( min( size_top, size_left ), nb_alg ) ) ) break;
```

The first call determines how many rows of the current view `ABR` can be split off while guaranteeing that those rows exist within one row on nodes. Similarly, the second call determines how many columns of the current view `ABR` can be split off while guaranteeing that those rows exist within one row on nodes. The size of A_{11} is then determined to be the smallest of these two values and the algorithmic block size.

Once it is guaranteed that A_{11} exists on one node, the call to `PLA_Chol_right_0` can be replaced by a call to `PLA_Local_chol` which performs a sequential Cholesky factorization on the node that owns A_{11}. Other than this, the code is essentially identical.

Note 4.3.1 *This code continues to have a clear shortcoming in that only the node that owns the current diagonal element α_{11} will determine that a square-root of a negative number is to be taken, and thus only that node will recognize an error condition.*

Note 4.3.2 *Fig.* **??** *shows that performance is now more acceptable impressive: the code attains better than half of peak for very large problem sizes.*

Note 4.3.3 *Expain subtlety in* `PLA_Obj_split_size` *functionality.*

```
#include "PLA.h"
#include "mpi.h"

Chol_right_2( PLA_Obj A, int nb_alg, int *info )
{
  int       size,                         size_top,
            size_left,                    dummy;
  PLA_Obj   ABR             = NULL,     A11             = NULL,
            A21             = NULL,
            one             = NULL,     minus_one       = NULL;

  /* Create usual duplicated scalar constants */
  PLA_Create_constants_conf_to( A, &minus_one, NULL, &one );

  /* View ABR = A */
  PLA_Obj_view_all( A, &ABR );

  while ( TRUE ) {
    /* Determine size of current panel */
    PLA_Obj_split_size( ABR, PLA_SIDE_LEFT, &size_left, &dummy );
    PLA_Obj_split_size( ABR, PLA_SIDE_TOP,  &size_top,  &dummy );
    if ( 0 == ( size = min( min( size_top, size_left ), nb_alg ) ) ) break;

    PLA_Obj_split_4( ABR, size, size,    &A11, PLA_DUMMY,
                                         &A21, &ABR );

     *info = PLA_Local_chol( PLA_LOWER_TRIANGULAR, A11 );
    if ( *info != 0 ) break;

    PLA_Trsm( PLA_SIDE_RIGHT, PLA_LOW_TRIAN, PLA_TRANS, PLA_NONUNIT_DIAG,
              one, A11, A21 );
    PLA_Syrk( PLA_LOW_TRIAN, PLA_NO_TRANS, minus_one, A21, one, ABR );
  }

  PLA_Obj_free( &ABR );        PLA_Obj_free( &A11 );
  PLA_Obj_free( &A21 );
  PLA_Obj_free( &one );        PLA_Obj_free( &minus_one );
}
```

Figure 4.6: Factoring A_{11} Local on One Node

4.4 Updating A_{21} on One Column of Nodes

A next optimization comes from the recognition that if A_{11} exists within one node, then A_{21} exists within one column of nodes. Since the update of individual rows of A_{21} is only dependent upon that row and A_{11}, it suffices to communication A_{11} to all nodes within the same column of nodes as the node that owns A_{11}, after which local triangular solves can commence to update A_{21}.

The first step is to communication A_{11} to all nodes in the same column of nodes. This is accomplished by first casting A11 to become a multiscalar (owned by one node):

```
        PLA_Obj_objtype_cast( A11, PLA_MSCALAR );
```

After this, a duplicated multiscalar that inherits all attributes of A11 except is duplicated to all rows within the column that owns A_{11} is created:

```
        PLA_Mscalar_create_conf_to( A11, PLA_ALL_ROWS, PLA_INHERIT, &A11_dup );
```

Finally, the duplication is accomplished by copying from A11 to A11_dup.

```
        PLA_Copy( A11, A11_dup );
```

Now all nodes in the appropriate column own the data required to perform the local triangular solve with multiple right-hand-sides:

```
        PLA_Local_trsm( PLA_SIDE_RIGHT, PLA_LOW_TRIAN, PLA_TRANS, PLA_NONUNIT_DIAG,
                        one, A11_dup, A21 );
```

The complete code is given in Fig. 4.7.

Note 4.4.1 *This code continues to have a clear shortcoming in that only the node that owns the current diagonal element α_{11} will determine that a square-root of a negative number is to be taken, and thus only that node will recognize an error condition.*

Note 4.4.2 *Fig. 4.8 shows that performance continues to improve. However, the improvement in not nearly as dramatic, since not much communication was being performed by the global* PLA_Trsm*.*

```
#include "PLA.h"
#include "mpi.h"

void Chol_right_3( PLA_Obj A, int nb_alg, int *info )
{
  int       size,                        size_top,
            size_left,                   dummy;
  PLA_Obj   ABR              = NULL,     A11              = NULL,
            A21              = NULL,     A11_dup          = NULL,
            one              = NULL,     minus_one        = NULL,
            zero             = NULL;

  /* Create usual duplicated scalar constants */
  PLA_Create_constants_conf_to( A, &minus_one, &zero, &one );

  /* View ABR = A */
  PLA_Obj_view_all( A, &ABR );

  while ( TRUE ) {
    /* Determine size of current panel */
    PLA_Obj_split_size( ABR, PLA_SIDE_LEFT, &size_left, &dummy );
    PLA_Obj_split_size( ABR, PLA_SIDE_TOP,  &size_top,  &dummy );
    if ( 0 == ( size = min( min( size_top, size_left ), nb_alg ) ) ) break;

    PLA_Obj_split_4( ABR, size, size,    &A11, PLA_DUMMY,
                                         &A21, &ABR );

    *info = PLA_Local_chol( PLA_LOWER_TRIANGULAR, A11 );
    if ( *info != 0 ) break;

    PLA_Obj_objtype_cast( A11, PLA_MSCALAR );
    PLA_Mscalar_create_conf_to( A11, PLA_ALL_ROWS, PLA_INHERIT, &A11_dup );
    PLA_Copy( A11, A11_dup );

    PLA_Local_trsm( PLA_SIDE_RIGHT, PLA_LOW_TRIAN, PLA_TRANS, PLA_NONUNIT_DIAG,
                    one, A11_dup, A21 );

    PLA_Syrk( PLA_LOW_TRIAN, PLA_NO_TRANS, minus_one, A21, one, ABR );
  }

  PLA_Obj_free( &ABR );        PLA_Obj_free( &A11 );
  PLA_Obj_free( &A21 );        PLA_Obj_free( &A11_dup );
  PLA_Obj_free( &one );        PLA_Obj_free( &minus_one );
}
```

Figure 4.7: Updating A_{21} on One Column of Nodes

4.5 Performance

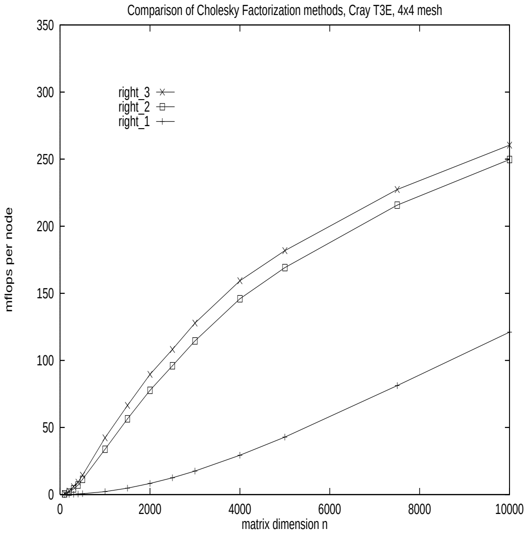


Figure 4.8: Performance of various PLAPACK implementations on a 16 processor configuration of the Cray T3E. A distribution block size of 64 and algorithmic block size of 128 was used to collect this data.

Chapter 5

PLAPACK Cholesky Factorization: You too can beat ScaLAPACK performance

Heck, I wanna fly

5.1 Factoring the Current Panel Using All Nodes

5.2 For Speed Fanatics Only

5.3 Performance

```
PLA_Chol_right_4( PLA_Obj A, int nb_alg, int *info )
{
  int       size;
  PLA_Obj   ABL             = NULL,     ABR             = NULL,
            A1              = NULL,     A1_mv           = NULL,
            A1_mv_cur       = NULL,     A21             = NULL,
            one             = NULL,     minus_one       = NULL;

  /* Create usual duplicated scalar constants */
  PLA_Create_constants_conf_to( A, &minus_one, NULL, &one );

  /* Initially partition A = / ATL    |    *   \  where
                             \ ABL    | A11_BR /  ATL is 0x0  */
  PLA_Obj_split_4( A, 0, 0,    PLA_DUMMY, PLA_DUMMY,
                               &ABL,      &ABR );

  /* Create a multivector to hold current panel */
  PLA_Mvector_create_conf_to( ABL, nb_alg, &A1_mv );

  while ( TRUE ) {
    /* Determine size of current panel */
    PLA_Obj_global_length( ABR, &size );
    if ( 0 == ( size = min( size, nb_alg ) ) ) break;

    /* Split off current panel */
    PLA_Obj_vert_split_2( ABR, size, &A1, PLA_DUMMY );
    /* Size the workspace for the multivector copy of A1 */
    PLA_Obj_vert_split_2( A1_mv, size,       &A1_mv_cur, PLA_DUMMY );

    /* Copy A1 to A1_mv, factor the current panel, and place back in A1 */
    PLA_Copy( A1, A1_mv_cur );
    PLA_Chol_mv( A1_mv_cur, info );
    PLA_Copy( A1_mv_cur, A1 );
    if ( *info != 0 ) break;

    PLA_Obj_split_4( ABR, size, size,    PLA_DUMMY, PLA_DUMMY,
                                         &A21,      &ABR );
    PLA_Syrk_new( PLA_LOW_TRIAN, PLA_NO_TRANS, minus_one, A21, one, ABR );

    /* Update view of A1_mv */
    PLA_Obj_horz_split_2( A1_mv, size,   PLA_DUMMY,
                                         &A1_mv );
  }

  PLA_Obj_free( &ABL );        PLA_Obj_free( &ABR );
  PLA_Obj_free( &A1 );         PLA_Obj_free( &A1_mv );
  PLA_Obj_free( &A1_mv_cur );  PLA_Obj_free( &A21 );
  PLA_Obj_free( &one );        PLA_Obj_free( &minus_one );
}
```

Figure 5.1: Factoring the Current Panel Using All Nodes

```
int PLA_Chol_mv( PLA_Obj Ap, int *info )
{
  int       size,            dummy;
  PLA_Obj   A11    = NULL,                       L11    = NULL,
            A21    = NULL,
            A11_BL = NULL,   A11_BR = NULL,      L11_BL = NULL,   L11_BR = NULL,
            A11_10 = NULL,   A11_11 = NULL,      L11_10 = NULL,   L11_11 = NULL,
            A11_20 = NULL,   A11_21 = NULL,
            A11_R1 = NULL,                       L11_R1 = NULL,
            one = NULL,      minus_one = NULL;

  PLA_Create_constants_conf_to( Ap, &minus_one, NULL, &one );

  PLA_Obj_global_width( Ap, &size );
  PLA_Obj_horz_split_2( Ap, size,        &A11,
                                         &A21 );

  PLA_Mscalar_create_conf_to( A11, PLA_ALL_ROWS, PLA_ALL_COLS, &L11 );

  PLA_Obj_split_4( A11, 0, 0,    PLA_DUMMY, PLA_DUMMY,
                                 &A11_BL,   &A11_BR    );
  PLA_Obj_split_4( L11, 0, 0,    PLA_DUMMY, PLA_DUMMY,
                                 &L11_BL,   &L11_BR    );

```

Figure 5.2: Factoring the Current Panel Using All Nodes.

```
  while ( TRUE ) {
    PLA_Obj_split_size( A11_BL, PLA_SIDE_TOP, &size, &dummy );
    PLA_Obj_horz_split_2( A11_BL, size,         &A11_10,
                                                &A11_20 );
    PLA_Obj_split_4     ( A11_BR, size, size,   &A11_11,  PLA_DUMMY,
                                                &A11_21,  &A11_BR );
    PLA_Obj_horz_split_2( L11_BL, size,         &L11_10,
                                                PLA_DUMMY );
    PLA_Obj_split_4     ( L11_BR, size, size,   &L11_11,   PLA_DUMMY,
                                                PLA_DUMMY, &L11_BR    );

    PLA_Local_syrk( PLA_LOW_TRIAN, PLA_NO_TRANS,
                    minus_one, A11_10, one, A11_11 );

    PLA_Obj_view_all( A11_10, &A11_R1 );
    PLA_Obj_view_shift( A11_R1,       0,
                                0,        size,
                                      0 );
    PLA_Obj_view_all( L11_10, &L11_R1 );
    PLA_Obj_view_shift( L11_R1,       0,
                                0,        size,
                                      0 );
    PLA_Obj_objtype_cast( A11_R1, PLA_MSCALAR );

    PLA_Copy( A11_R1, L11_R1 );

    *info = PLA_Local_chol( L11_11 );

    PLA_Local_copy( L11_11, A11_11 );

    if ( *info != 0 ) break;

    PLA_Local_gemm( PLA_NO_TRANS, PLA_TRANS, minus_one, A11_20, L11_10, one, A11_21 );
    PLA_Local_trsm( PLA_SIDE_RIGHT, PLA_LOW_TRIAN, PLA_TRANS, PLA_NONUNIT_DIAG,
                    one, L11_11, A11_21 );

    PLA_Obj_view_shift( A11_BL,         size,
                                  0,          size,
                                         0 );
    PLA_Obj_view_shift( L11_BL,         size,
                                  0,          size,
                                         0 );
  }

  if ( *info == 0 )
    PLA_Local_trsm( PLA_SIDE_RIGHT, PLA_LOW_TRIAN, PLA_TRANS, PLA_NONUNIT_DIAG,
                    one, L11, A21 );

  < free temporary views >
  return( PLA_SUCCESS );
}
```

Figure 5.3: Factoring the Current Panel Using All Nodes (continued.

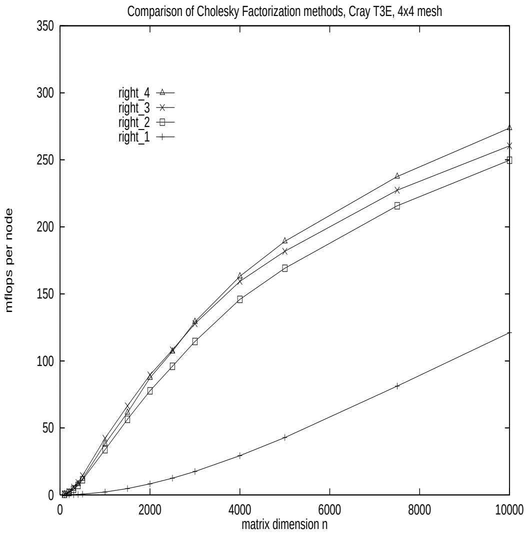


Figure 5.4: Performance of ??? PLAPACK implementations on a 16 processor configuration of the Cray T3E. A distribution block size of 64 and algorithmic block size of 128 was used to collect this data.

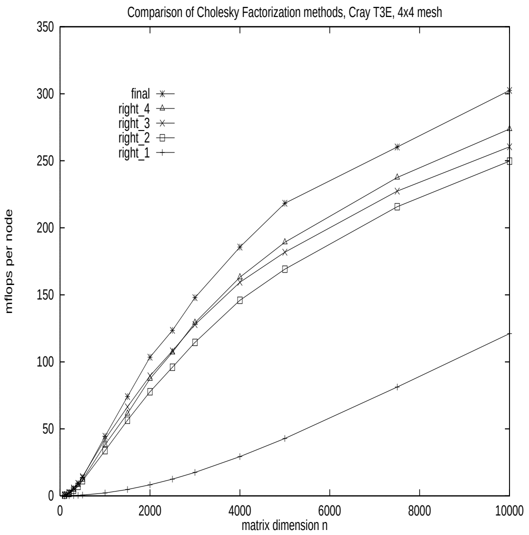


Figure 5.5: Performance of ??? PLAPACK implementations on a 16 processor configuration of the Cray T3E. A distribution block size of 64 and algorithmic block size of 128 was used to collect this data.