Zen and the Art of High Performance Parallel Computing

C Version

by
Robert A. van de Geijn

September 18, 1998

Contents

1 Introduction **2**

2 Hey, I Just Want to Solve My $#%#$ Problem **3**
2.1 A simple driver 3
2.2 Building a problem 9
2.3 Getting the Answer 13
2.4 Putting it all together 13

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.

```
1   #include "mpi.h"
2   #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

```
6       MPI_Comm comm;
```

```
1   #include "mpi.h"
2   #include "PLA.h"
3
4   int main(int argc, char *argv[])
5   {
6     MPI_Comm comm;
7     PLA_Template templ = NULL;
8     PLA_Obj  A = NULL,           x = NULL,         b = NULL,
9              minus_one = NULL,   b_norm = NULL;
10    double   *locA, b_norm_value, time;
11    int      size, nb_distr, me, nprocs, dummy, info = 0;
12
13    MPI_Init(&argc, &argv);
14
15    PLA_Comm_1D_to_2D_ratio(MPI_COMM_WORLD, 1.0, &comm);
16    PLA_Init(comm);
17
18    MPI_Comm_rank(MPI_COMM_WORLD, &me);
19    if (0 == me) {
20      printf("enter matrix size and distribution block size:\n");
21      scanf("%d %d", &size, &nb_distr );
22    }
23
24    MPI_Bcast(&size,     1, MPI_INT, 0, MPI_COMM_WORLD);
25    MPI_Bcast(&nb_distr, 1, MPI_INT, 0, MPI_COMM_WORLD);
26
27    PLA_Temp_create( nb_distr, 0, &templ );
28
29    PLA_Matrix_create(  MPI_DOUBLE, size, size,
30                        templ, PLA_ALIGN_FIRST, PLA_ALIGN_FIRST, &A );
31
32    PLA_Mvector_create( MPI_DOUBLE, size, 1, templ, PLA_ALIGN_FIRST, &x );
33
34    PLA_Mvector_create( MPI_DOUBLE, size, 1, templ, PLA_ALIGN_FIRST, &b );
35
36    PLA_Create_constants_conf_to( A, &minus_one, NULL, NULL );
37
38    create_problem( A, x, b );
39
40    MPI_Barrier( MPI_COMM_WORLD );
41    time = MPI_Wtime( );
42
43    info = PLA_Chol( PLA_LOWER_TRIANGULAR, A );
44
45    if ( info != 0 ) {
46      printf("nonpositive definite matrix detected\n");
47    }
```

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

```
48    else {
49      PLA_Trsv( PLA_LOW_TRIAN, PLA_NO_TRANS, PLA_NONUNIT_DIAG, A, b );
50      PLA_Trsv( PLA_LOW_TRIAN, PLA_TRANS,    PLA_NONUNIT_DIAG, A, b );
51
52      MPI_Barrier( MPI_COMM_WORLD );
53      time = MPI_Wtime() - time;
54
55      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
56      if ( me == 0 )
57        printf("n = %d, time = %f, MFLOPS/node = %f\n", size, time,
58               1.0/3.0 * size * size * size / time * 1.0e-6 / nprocs );
59
60      process_answer( b );
61
62      PLA_Axpy( minus_one, x, b );
63
64      PLA_Mscalar_create( MPI_DOUBLE, PLA_ALL_ROWS, PLA_ALL_COLS,
65                          1, 1, templ, &b_norm );
66
67      PLA_Nrm2( b, b_norm );
68
69      if ( me == 0 ) {
70        PLA_Obj_get_local_contents( b_norm, PLA_NO_TRANS, &dummy, &dummy,
71                                    &b_norm_value, 1, 1 );
72        printf( "Norm of difference between original x and computed x: %e \n",
73                b_norm_value );
74      }
75    }
76
77    PLA_Obj_free(&A);            PLA_Obj_free(&x);
78    PLA_Obj_free(&b);            PLA_Obj_free(&minus_one);
79    PLA_Obj_free(&b_norm);
80
81    PLA_Temp_free(&templ);
82    PLA_Finalize( );
83    MPI_Finalize( );
84  }
```

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

```
7     PLA_Template templ = NULL;
8     PLA_Obj  A = NULL,           x = NULL,         b = NULL,
9              minus_one = NULL,   b_norm = NULL;
10    double   *locA, b_norm_value, time;
11    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.

```
13    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.

```
15    PLA_Comm_1D_to_2D_ratio(MPI_COMM_WORLD, 1.0, &comm);
16    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.

```
19    MPI_Comm_rank(MPI_COMM_WORLD, &me);
20    if (0 == me) {
21      printf("enter matrix size and distribution block size:\n");
22      scanf("%d %d", &size, &nb_distr );
23    }
24
25    MPI_Bcast(&size,     1, MPI_INT, 0, MPI_COMM_WORLD);
26    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.

```
27    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.

```
29    PLA_Matrix_create(  MPI_DOUBLE, size, size,
30                        templ, PLA_ALIGN_FIRST, PLA_ALIGN_FIRST, &A );
31
32    PLA_Mvector_create( MPI_DOUBLE, size, 1, templ, PLA_ALIGN_FIRST, &x );
33
34    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.

```
36    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

```
38    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.

```
40    MPI_Barrier( MPI_COMM_WORLD );
41    time = MPI_Wtime( );
42
43    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.

```
45      if ( info != 0 ) {
46        printf("nonpositive definite matrix detected\n");
47      }
48      else {
49        PLA_Trsv( PLA_LOW_TRIAN, PLA_NO_TRANS, PLA_NONUNIT_DIAG, A, b );
50        PLA_Trsv( PLA_LOW_TRIAN, PLA_TRANS,    PLA_NONUNIT_DIAG, A, b );
51
52        MPI_Barrier( MPI_COMM_WORLD );
53        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

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

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

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

```
55        MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
56        if ( me == 0 ) {
57          MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
58          printf("n = %d, time = %f, MFLOPS/node = %f\n", size, time,
59                 1.0/3.0 * size * size * size / time * 1.0e-6 / nprocs );
60        }
```

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.

```
60        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 $||b - x||_2$.

```
62        PLA_Axpy( minus_one, x, b );
63
64        PLA_Mscalar_create( MPI_DOUBLE, PLA_ALL_ROWS, PLA_ALL_COLS,
65                            1, 1, templ, &b_norm );
66
67        PLA_Nrm2( b, b_norm );
```

```
68
69      if ( me == 0 ) {
70        PLA_Obj_get_local_contents( b_norm, PLA_NO_TRANS, &dummy, &dummy,
71                                    &b_norm_value, 1, 1 );
72        printf( "Norm of difference between original x and computed x: %e \n",
73                 b_norm_value );
74      }
75    }
```

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.

```
77    /* Free to linear algebra objects */
78    PLA_Obj_free(&A);            PLA_Obj_free(&x);
79    PLA_Obj_free(&b);            PLA_Obj_free(&minus_one);
80    PLA_Obj_free(&b_norm);
81
82    /* Free the template */
83    PLA_Temp_free(&templ);
84
85    /* Finalize PLAPACK and MPI */
86    PLA_Finalize( );
87    MPI_Finalize( );
88  }
```

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 $Ax = b$.

```
4   void create_problem( PLA_Obj A, PLA_Obj x, PLA_Obj b )
5   {
```

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.

```
10    PLA_Obj_global_length( A, &size );              /* Extract dimension of A */
11
12    MPI_Comm_rank( MPI_COMM_WORLD, &me );
13    MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
```

```
1   #include "mpi.h"
2   #include "PLA.h"
3
4   void create_problem( PLA_Obj A, PLA_Obj x, PLA_Obj b )
5   {
6     PLA_Obj  zero = NULL,    one = NULL;
7     int size, me, nprocs, i, j;
8     double *locA, d_one = 1.0;
9
10    PLA_Obj_global_length( A, &size );
11
12    MPI_Comm_rank( MPI_COMM_WORLD, &me );
13    MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
14
15    PLA_Create_constants_conf_to( A, NULL, &zero, &one );
16
17    PLA_Obj_set_to_zero ( A );        PLA_Obj_set_to_zero ( x );
18    PLA_Obj_set_to_zero ( b );
19
20    PLA_API_begin();
21      PLA_Obj_API_open(A);            PLA_Obj_API_open(x);
22
23      srand48( me * 1793 );
24
25      locA = (double *) malloc( sizeof( double ) * size  );
26      for (j=me;j< size; j+=nprocs) {
27        for (i=0; i < size; i++) locA[i]=drand48() * 2.0 - 1.0;
28        locA[j] += size;
29        PLA_API_axpy_matrix_to_global( size, 1, &d_one, locA, size, A, 0, j );
30      }
31
32      if (0==me) {
33        for (i=0; i<size; i++) locA[i]=drand48() * 2.0 - 1.0;
34        PLA_API_axpy_vector_to_global( size, &d_one, locA, 1, x, 0);
35      }
36      free( locA );
37
38      PLA_Obj_API_close(x);           PLA_Obj_API_close(A);
39    PLA_API_end();
40
41    PLA_Symv( PLA_LOW_TRIAN, one, A, x, zero, b );
42
43    PLA_Obj_free( &zero );         PLA_Obj_free( &one );
44  }
```

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.

```
17    PLA_Obj_set_to_zero ( A );          PLA_Obj_set_to_zero ( x );
18    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`.

```
20    PLA_API_begin();
```

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

```
39     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( )`.

```
21      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()`.

```
38       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`.

```
23      srand48( me * 1793 );
24
25      locA = (double *) malloc( sizeof( double ) * size  );
26      for (j=me;j< size; j+=nprocs) {
27        for (i=0; i < size; i++) locA[i]=drand48() * 2.0 - 1.0;
28        locA[j] += size;
29        PLA_API_axpy_matrix_to_global( size, 1, &d_one, locA, size, A, 0, j );
30      }
```

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` .

```
32      if (0==me) {
33        for (i=0; i<size; i++) locA[i]=drand48() * 2.0 - 1.0;
34        PLA_API_axpy_vector_to_global( size, &d_one, locA, 1, x, 0);
35      }
36      free( locA );
```

Computing $b = Ax$

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

Clean up the environment

```
43    PLA_Obj_free( &zero );          PLA_Obj_free( &one );
44  }
```

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
```

```
1   #include "PLA.h"
2
3   void process_answer( PLA_Obj x )
4   {
5     int size, me, i;
6     double *locx, d_one = 1.0;
7
8     MPI_Comm_rank( MPI_COMM_WORLD, &me );
9
10    PLA_API_begin();
11      PLA_Obj_API_open( x );
12
13      if ( me == 0 ) {
14        PLA_Obj_global_length( x, &size );
15        locx = (double *) malloc( sizeof( double ) * size  );
16        for (i=0; i<size; i++) locx[i] = 0.0;
17
18        PLA_API_axpy_global_to_vector( size, &d_one, x, 0, locx, 1);
19      }
20
21      PLA_Obj_API_sync( x );
22
23      if ( me == 0 ) {
24        printf("first computed entry: %f\n", locx[0] );
25        printf("last  computed entry: %f\n", locx[size-1] );
26
27        free( locx );
28      }
29
30      PLA_Obj_API_close(x);
31    PLA_API_end();
32  }
```

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.