Zen and the Art of High Performance Parallel Computing

FORTRAN Version for the Cray T3D/E

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 8
2.3 Getting the Answer 11
2.4 Putting it all together 11

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. 2.1–2.4 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.

```
5          include "mpif.h"
6          include "PLAf.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 (integer that uniquely identifies it). By giving this handle to inquiry routines, information about the matrices or vectors can be extracted.

We will discuss declaration and initialization of objects as the are used.

Initialize MPI

A simple call initialized MPI.

```
54         CALL MPI_INIT( IERROR )
```

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. Consider the statements

```
10         INTEGER COMM

27         COMM = MPI_COMM_NULL

41         CALL PLA_COMM_1D_TO_2D_RATIO_F ( MPI_COMM_WORLD, 1.0E00, COMM,
42        &     IERROR )
43         CALL PLA_INIT_F( COMM, IERROR )
```

A communicator is an opaque object that contains all information that describes the processors (nodes) being used. In Fortran, a variable that will access a communicator is declared as an integer, and initialized to `MPI_COMM_NULL`, a constant declared in `mpif.h`. The group of all processors being used is given by the communicator `MPI_COMM_WORLD`. The routine

```
PLA_COMM_1D_TO_2D_RATIO_F( COMM_1D, RATIO, COMM_2D )
```

creates a 2D communicator `COMM_2D` from 1D communicator `COMM_1D` with a ratio of rows to columns of approximately `RATIO` (a 64 bit floating point number). Thus, the above statements initializes PLAPACK with a 2D communicator for a mesh of nodes that is roughly square.

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 $8, 16, 24, 32, 48, 64$.

After processor with rank 0 reads the parameters, they are broadcast to all processors.

```
47         CALL MPI_COMM_RANK( MPI_COMM_WORLD, ME, IERROR )
48         if ( ME .EQ. 0 ) THEN
49            PRINT *, "-- enter matrix size, distr. block size:"
50            READ  *, SIZE, NB_DISTR
51         ENDIF
52
53         CALL MPI_BCAST( SIZE,     1, MPI_INTEGER, 0, MPI_COMM_WORLD,
54        &     IERROR )
55         CALL MPI_BCAST( NB_DISTR, 1, MPI_INTEGER, 0, MPI_COMM_WORLD,
56        &     IERROR )
```

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_F` 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 third parameter. Thus the following statements declare and initialize a template.

```
14        INTEGER TEMPL

28        TEMPL = 0

60        CALL PLA_TEMP_CREATE_F( NB_DISTR, 0, TEMPL, IERROR )
```

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.

```
18        INTEGER A, X, B, B_NORM, MINUS_ONE, ZERO, ONE

25        INTEGER DATATYPE

29        A = 0
30        X = 0
31        B = 0
32        B_NORM = 0
33        MINUS_ONE = 0
34        ZERO = 0
35        ONE = 0

64        DATATYPE = MPI_REAL

68        CALL PLA_MATRIX_CREATE_F(  DATATYPE, SIZE, SIZE, TEMPL,
69       &                           PLA_ALIGN_FIRST, PLA_ALIGN_FIRST,
70       &                           A, IERROR )
71
72        CALL PLA_MVECTOR_CREATE_F( DATATYPE, SIZE, 1,
73       &                           TEMPL, PLA_ALIGN_FIRST,
74       &                           X, IERROR )
75
76        CALL PLA_MVECTOR_CREATE_F( DATATYPE, SIZE, 1,
77       &                           TEMPL, PLA_ALIGN_FIRST,
78       &                           B, IERROR )
```

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.

```
18          INTEGER  ... , MINUS_ONE, ZERO, ONE

82          CALL PLA_CREATE_CONSTANTS_CONF_TO_F( A, MINUS_ONE, ZERO, ONE,
83         &     IERROR )
```

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

```
87          CALL CREATE_PROBLEM_F( 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_F 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.

```
91          CALL MPI_BARRIER( MPI_COMM_WORLD, IERROR )
92          TIME = MPI_WTIME( )

96          CALL PLA_CHOL_F( PLA_LOWER_TRIANGULAR, A, INFO );
```

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_F. It is the Cholesky factorization together with the triangular solves that are of interest, and thus we stop timing after these solves.

```
98          IF ( INFO .NE. 0 ) THEN
99             PRINT *, "Nonpositive definite matrix detected"
100         ELSE

104            CALL PLA_TRSV_F( PLA_LOWER_TRIANGULAR, PLA_NO_TRANSPOSE,
105        &                    PLA_NONUNIT_DIAG, A, B, IERROR )

109            CALL PLA_TRSV_F( PLA_LOWER_TRIANGULAR, PLA_TRANSPOSE,
110        &                    PLA_NONUNIT_DIAG, A, B, IERROR )

114            CALL MPI_BARRIER( MPI_COMM_WORLD, IERROR )
115            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

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

```
119           IF ( ME .EQ. 0 ) THEN
120              CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROCS, IERROR )
121
122              OPERATION_COUNT = 1.0/3.0 * SIZE * SIZE * SIZE
123
124              PRINT *, "N =", SIZE, "TIME =", TIME, "MFLOPS/node =",
125       &           OPERATION_COUNT / TIME * 1.0e-6 / NPROCS
126           ENDIF
```

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_F discussed in detail in Section 2.3.

```
132           CALL PROCESS_ANSWER_F( B )
```

Check accuracy

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

```
137           CALL PLA_AXPY_F( MINUS_ONE, X, B, IERROR )

141           CALL PLA_MSCALAR_CREATE_F( DATATYPE,
142       &                              PLA_ALL_ROWS, PLA_ALL_COLS,
143       &                              1, 1, TEMPL, B_NORM, IERROR )

145           CALL PLA_NRM2_F ( B, B_NORM, IERROR )

149           IF ( ME .EQ. 0 ) THEN
150              CALL PLA_OBJ_GET_LOCAL_CONTENTS_F(
151       &           B_NORM, PLA_NO_TRANS, DUMMY, DUMMY,
152       &           B_NORM_VALUE, 1, 1, IERROR )
153              PRINT *, "||(computed x) - x ||_2 = ", B_NORM_VALUE
154           ENDIF
155        ENDIF
```

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.

```
159         CALL PLA_OBJ_FREE_F( A, IERROR )
160         CALL PLA_OBJ_FREE_F( X, IERROR )
161         CALL PLA_OBJ_FREE_F( B, IERROR )
162         CALL PLA_OBJ_FREE_F( B_NORM, IERROR )
163         CALL PLA_OBJ_FREE_F( MINUS_ONE, IERROR )
164         CALL PLA_OBJ_FREE_F( ZERO, IERROR )
165         CALL PLA_OBJ_FREE_F( ONE, IERROR )

169         CALL PLA_TEMP_FREE_F( TEMPL, IERROR )

173         CALL PLA_FINALIZE_F(IERROR )
174         CALL MPI_FINALIZE( IERROR )
```

2.2 Building a problem

The sample program given in Fig. 2.5–2.7 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 $Ax = b$.

```
1           SUBROUTINE CREATE_PROBLEM_F( A, X, 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. Since we use static allocation for the local arrays that will be used to fill the matrix and vectors, we prescribe a maximum size of the problem that can be solved, `MAX_SIZE`, which can be changed in the declarations of this routine.

```
41          CALL PLA_OBJ_GLOBAL_LENGTH_F( A, SIZE, IERROR )
42          IF ( SIZE .GT. MAX_SIZE ) THEN
43             PRINT *, "SIZE > MAX_SIZE IN CREATE_PROBLEM_F"
44             STOP
45          ENDIF
46
47          CALL MPI_COMM_RANK( MPI_COMM_WORLD, ME, IERROR )
48          CALL MPI_COMM_SIZE( MPI_COMM_WORLD, NPROCS, IERROR )
```

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, x, and b to equal zero.

```
50          CALL PLA_OBJ_SET_TO_ZERO_F( A, IERROR )
51          CALL PLA_OBJ_SET_TO_ZERO_F( X, IERROR )
52          CALL PLA_OBJ_SET_TO_ZERO_F( B, IERROR )
```

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_F`.

```
58          CALL PLA_API_BEGIN_F( IERROR )
```

Upon completion, we leave the API mode by calling `PLA_API_END_F`.

```
110         CALL PLA_API_END_F( IERROR )
```

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_F`.

```
62              CALL PLA_OBJ_API_OPEN_F( A, IERROR )
63              CALL PLA_OBJ_API_OPEN_F( X, IERROR )
```

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_F()`.

```
105         CALL PLA_OBJ_API_CLOSE_F( X, IERROR )
106         CALL PLA_OBJ_API_CLOSE_F( A, IERROR )
```

Caution: *Contributions to a linear algebra object can NOT assumed to actually have been written until a call to* `PLA_OBJ_API_CLOSE_F` *or* `PLA_OBJ_API_SYNC_F` *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:

$$A = \left(\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_n \end{array} \right)$$

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 $(i-1) \bmod p$ 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_F`, with calling sequence

```
PLA_API_AXPY_MATRIX_TO_GLOBAL( M, N, ALPHA, A_LOCAL, LDA_LOCAL,
                               A_GLOBAL, I, J )
```

where M, and N are the dimension of the local matrix being submitted, ALPHA is a scaling factor, A_LOCAL is the address where the local matrix starts, with local leading dimension LDA_LOCAL. The global matrix to which this contribution is to be added is given by A_GLOBAL, an object handle, and I and J given the index of the upper-right-hand corner of the submatrix of the global matrix to which the local contribution is to be added. Thus, the operation performed (using MATLAB-like syntax) is

```
A_GLOBAL( I:I+M-1,J:J+N-1 ) =
             ALPHA * A_LOCAL + A_GLOBAL( I:I+M-1,J:J+N-1 );
```

Notice: whether indexing starts at 0 or 1 for the local array is completely independent of PLAPACK. Whether the global indexing (used for A_GLOBAL) starts at 0 or 1 depends on how the template was created for object A_GLOBAL.

Contributions from each processor will be generated using a random number generator. To ensure that the random numbers on each processor are not repeated on other processors, we seed the random number generator differently on each processor.

```
67            CALL SEED_RANDOM( ME * 1793 )
```

The following code generates columns ME+1, ME+NPROCS+1, ME+2*NPROCS+1, ... on processor with index ME.

```
71            DO J=ME+1, SIZE, NPROCS


89            ENDDO
```

Within this loop, in local array Z_COL we generate a column of random numbers. By adding the matrix size to the Jth element, we ensure that the matrix is diagonally dominant, and thus positive definite.

```
75               DO I=1, SIZE
76                  IF ( MPI_REAL .EQ. DATATYPE ) THEN
77                     CALL REAL_RANDOM( Z_COL( I ) )
78                  ELSE
79                     PRINT *, "DATATYPE UNKNOWN"
80                     STOP
81                  ENDIF
82               ENDDO
83               Z_COL( J ) = Z_COL( J ) + SIZE
```

Once the column is created, it is submitted to addition to the global matrix A:

```
87               CALL PLA_API_AXPY_MATRIX_TO_GLOBAL_F(
88        &            SIZE, 1, 1.0E00, Z_COL, SIZE, A, 0, J-1, IERROR )
89
```

Notice: *a call to* PLA_API_AXPY_MATRIX_TO_GLOBAL_F *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_F *or* PLA_OBJ_API_SYNC_F *has been made.*

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

```
90            IF ( ME .EQ. 0 ) THEN
91               DO I=1, SIZE
92                  IF ( MPI_REAL .EQ. DATATYPE ) THEN
93                     CALL REAL_RANDOM( Z_COL( I ) )
94                  ELSE
95                     PRINT *, "DATATYPE UNKNOWN"
96                     STOP
97                  ENDIF
98               ENDDO
99               CALL PLA_API_AXPY_VECTOR_TO_GLOBAL_F(
100      &            SIZE, 1.0E00, Z_COL( 1 ), 1, X, 0, IERROR )
101           ENDIF
```

Computing $b = Ax$

```
114       CALL PLA_CREATE_CONSTANTS_CONF_TO_F( A, MINUS_ONE, ZERO, ONE,
115      &     IERROR )
116       CALL PLA_SYMV_F( PLA_LOWER_TRIANGULAR, ONE, A, X, ZERO, B,
117      &     IERROR )
```

Clean up the environment

```
121       CALL PLA_OBJ_FREE_F( MINUS_ONE, IERROR )
122       CALL PLA_OBJ_FREE_F( ZERO, IERROR )
123       CALL PLA_OBJ_FREE_F( ONE, IERROR )
```

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. 2.8–2.9 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
```

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.

Installing the FORTRAN interface

At this moment in time, the FORTRAN interface is *not* part of the basic release. The best way to get the FORTRAN interface is to install the example given below.

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/FORTRAN_T3E/Chapter2
~/PLAPACK/Tutorial/chol/FORTRAN_T3E/Chapter2 % make
~/PLAPACK/Tutorial/chol/FORTRAN_T3E/Chapter2 % mpprun -n 4 Chol_driver.x
 -- enter matrix size, distr. block size:
4000 24
 N = 4000 TIME = 22.329607248306274 MFLOPS/node = 238.84581909687964
 First computed entry: -0.9227325475840108
 Last  computed entry: -0.24082651448829032
 ||(computed x) - x ||_2 =  2.30562461007449614E-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.

```
1          PROGRAM MAIN
2
3          implicit none
4
5          include "mpif.h"
6          include "PLAf.h"
7     C
8     C MPI_Comm
9     C
10         INTEGER COMM
11    C
12    C PLA_Template
13    C
14         INTEGER TEMPL
15    C
16    C PLA_Obj
17    C
18         INTEGER A, X, B, B_NORM, MINUS_ONE, ZERO, ONE
19    C
20         REAL OPERATION_COUNT, B_NORM_VALUE, TIME
21         INTEGER IERROR, SIZE, NB_DISTR, ME, NPROCS, DUMMY, INFO
22    C
23    C MPI_Datatype
24    C
25         INTEGER DATATYPE
26
27         COMM = MPI_COMM_NULL
28         TEMPL = 0
29         A = 0
30         X = 0
31         B = 0
32         B_NORM = 0
33         MINUS_ONE = 0
34         ZERO = 0
35         ONE = 0
36
37         CALL MPI_INIT( IERROR )
38    C
39    C Initialize PLAPACK
40    C
41         CALL PLA_COMM_1D_TO_2D_RATIO_F ( MPI_COMM_WORLD, 1.0E00, COMM,
42        &     IERROR )
43         CALL PLA_INIT_F( COMM, IERROR )
```

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

```
C
C Get problem size and distribution block size and broadcast
C
      CALL MPI_COMM_RANK( MPI_COMM_WORLD, ME, IERROR )
      if ( ME .EQ. 0 ) THEN
         PRINT *, "-- enter matrix size, distr. block size:"
         READ  *, SIZE, NB_DISTR
      ENDIF

      CALL MPI_BCAST( SIZE,     1, MPI_INTEGER, 0, MPI_COMM_WORLD,
     &     IERROR )
      CALL MPI_BCAST( NB_DISTR, 1, MPI_INTEGER, 0, MPI_COMM_WORLD,
     &     IERROR )
C
C Create object distribution template
C
      CALL PLA_TEMP_CREATE_F( NB_DISTR, 0, TEMPL, IERROR )
C
C Set the datatype : MPI_REAL   ON CRAY
C Set the datatype : MPI_DOUBLE_PRECISION O/W
      DATATYPE = MPI_REAL
C
C Create objects for problem to be solved
C
      CALL PLA_MATRIX_CREATE_F(  DATATYPE, SIZE, SIZE, TEMPL,
     &                           PLA_ALIGN_FIRST, PLA_ALIGN_FIRST,
     &                           A, IERROR )

      CALL PLA_MVECTOR_CREATE_F( DATATYPE, SIZE, 1,
     &                           TEMPL, PLA_ALIGN_FIRST,
     &                           X, IERROR )

      CALL PLA_MVECTOR_CREATE_F( DATATYPE, SIZE, 1,
     &                           TEMPL, PLA_ALIGN_FIRST,
     &                           B, IERROR )
C
C Create duplicated scalar constants with same datatype and template as A
C
      CALL PLA_CREATE_CONSTANTS_CONF_TO_F( A, MINUS_ONE, ZERO, ONE,
     &     IERROR )
C
C Create a problem to be solved: A x = b
C
      CALL CREATE_PROBLEM_F( A, X, B )
```

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

```
88  C
89  C Start timing
90  C
91        CALL MPI_BARRIER( MPI_COMM_WORLD, IERROR )
92        TIME = MPI_WTIME( )
93  C
94  C Factor A -> L L^T overwriting lower triangular portion of A with L
95  C
96        CALL PLA_CHOL_F( PLA_LOWER_TRIANGULAR, A, INFO );
97
98        IF ( INFO .NE. 0 ) THEN
99           PRINT *, "Nonpositive definite matrix detected"
100       ELSE
101 C
102 C Solve L y = b, overwriting b with y
103 C
104          CALL PLA_TRSV_F( PLA_LOWER_TRIANGULAR, PLA_NO_TRANSPOSE,
105     &                     PLA_NONUNIT_DIAG, A, B, IERROR )
106 C
107 C Solve trans(L) x = y (=b), overwriting b with x
108 C
109          CALL PLA_TRSV_F( PLA_LOWER_TRIANGULAR, PLA_TRANSPOSE,
110     &                     PLA_NONUNIT_DIAG, A, B, IERROR )
111 C
112 C Stop timing
113 C
114          CALL MPI_BARRIER( MPI_COMM_WORLD, IERROR )
115          TIME = MPI_WTIME() - TIME
116 C
117 C Report performance
118 C
119          IF ( ME .EQ. 0 ) THEN
120             CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROCS, IERROR )
121
122             OPERATION_COUNT = 1.0/3.0 * SIZE * SIZE * SIZE
123
124             PRINT *, "N =", SIZE, "TIME =", TIME, "MFLOPS/node =",
125     &           OPERATION_COUNT / TIME * 1.0e-6 / NPROCS
126          ENDIF
127 C
128 C Process the answer.  As an example, this routine brings
129 C result x (stored in b) to processor 0 and prints first and
130 C last entry
131 C
132          CALL PROCESS_ANSWER_F( B )
```

Figure 2.3: A simple driver routine for testing the Cholesky factorization, Part 3.

```
C
C Check answer by overwriting b <- b - x (where b holds computed
C approximation to x)
C
         CALL PLA_AXPY_F( MINUS_ONE, X, B, IERROR )
C
C Create 1x1 multiscalars to hold 2-norm of b - x
C
         CALL PLA_MSCALAR_CREATE_F( DATATYPE,
     &                              PLA_ALL_ROWS, PLA_ALL_COLS,
     &                              1, 1, TEMPL, B_NORM, IERROR )
C
         CALL PLA_NRM2_F ( B, B_NORM, IERROR )
C
C Report norm of b - x
C
         IF ( ME .EQ. 0 ) THEN
            CALL PLA_OBJ_GET_LOCAL_CONTENTS_F(
     &           B_NORM, PLA_NO_TRANS, DUMMY, DUMMY,
     &           B_NORM_VALUE, 1, 1, IERROR )
            PRINT *, "||(computed x) - x ||_2 = ", B_NORM_VALUE
         ENDIF
       ENDIF
C
C Free the linear algebra objects
C
       CALL PLA_OBJ_FREE_F( A, IERROR )
       CALL PLA_OBJ_FREE_F( X, IERROR )
       CALL PLA_OBJ_FREE_F( B, IERROR )
       CALL PLA_OBJ_FREE_F( B_NORM, IERROR )
       CALL PLA_OBJ_FREE_F( MINUS_ONE, IERROR )
       CALL PLA_OBJ_FREE_F( ZERO, IERROR )
       CALL PLA_OBJ_FREE_F( ONE, IERROR )
C
C Free the template
C
       CALL PLA_TEMP_FREE_F( TEMPL, IERROR )
C
C Finalize PLAPACK and MPI
C
       CALL PLA_FINALIZE_F(IERROR )
       CALL MPI_FINALIZE( IERROR )

       STOP
       END
```

Figure 2.4: A simple driver routine for testing the Cholesky factorization, Part 4.

```
1          SUBROUTINE CREATE_PROBLEM_F( A, X, B )
2     C
3     C Compute random A and x and let b = A x.
4     C
5     C
6          IMPLICIT NONE
7
8          INCLUDE "mpif.h"
9          INCLUDE "PLAf.h"
10    C
11    C PLA_Obj Parameters
12    C
13         INTEGER A, X, B
14    C
15    C PLA_Obj Local
16    C
17         INTEGER MINUS_ONE, ZERO, ONE
18    C
19    C Maximum matrix dimension (length of column)
20    C
21         INTEGER MAX_SIZE
22         PARAMETER ( MAX_SIZE = 10000 )
23    C
24         INTEGER SIZE, ME, NPROCS, I, J, IERROR
25         REAL Z_COL( MAX_SIZE )
26    C
27    C MPI_Datatype
28    C
29         INTEGER DATATYPE
30    C
31    C Initialize objects to NULL
32    C
33         MINUS_ONE = 0
34         ZERO = 0
35         ONE  = 0
36
37
38    C
39    C Extract dimension of A
40    C
41         CALL PLA_OBJ_GLOBAL_LENGTH_F( A, SIZE, IERROR )
42         IF ( SIZE .GT. MAX_SIZE ) THEN
43            PRINT *, "SIZE > MAX_SIZE IN CREATE_PROBLEM_F"
44            STOP
45         ENDIF
46
```

Figure 2.5: A sample subroutine for creating a symmetric positive definite matrix A and vectors x and b s.t. $Ax = b$.

```
47          CALL MPI_COMM_RANK( MPI_COMM_WORLD, ME, IERROR )
48          CALL MPI_COMM_SIZE( MPI_COMM_WORLD, NPROCS, IERROR )
49
50          CALL PLA_OBJ_SET_TO_ZERO_F( A, IERROR )
51          CALL PLA_OBJ_SET_TO_ZERO_F( X, IERROR )
52          CALL PLA_OBJ_SET_TO_ZERO_F( B, IERROR )
53
54          CALL PLA_OBJ_DATATYPE_F( A, DATATYPE, IERROR )
55    C
56    C Enter Application Interface mode
57    C
58          CALL PLA_API_BEGIN_F( IERROR )
59    C
60    C Open objects A and x for read/write
61    C
62             CALL PLA_OBJ_API_OPEN_F( A, IERROR )
63             CALL PLA_OBJ_API_OPEN_F( X, IERROR )
64    C
65    C Seed the random number generator
66    C
67             CALL SEED_RANDOM( ME * 1793 )
68    C
69    C Columns are computed by processors in a round-robin fashion
70    C
71             DO J=ME+1, SIZE, NPROCS
72    C
73    C Fill column j with random numbers
74    C
75                DO I=1, SIZE
76                   IF ( MPI_REAL .EQ. DATATYPE ) THEN
77                      CALL REAL_RANDOM( Z_COL( I ) )
78                   ELSE
79                      PRINT *, "DATATYPE UNKNOWN"
80                      STOP
81                   ENDIF
82                ENDDO
83                Z_COL( J ) = Z_COL( J ) + SIZE
84    C
85    C Submit the column to the global matrix
86    C
87                CALL PLA_API_AXPY_MATRIX_TO_GLOBAL_F(
88           &            SIZE, 1, 1.0E00, Z_COL, SIZE, A, 0, J-1, IERROR )
89             ENDDO
```

Figure 2.6: A sample subroutine for creating a symmetric positive definite matrix A and vectors x and b s.t. $Ax = b$, Part 2.

```
90            IF ( ME .EQ. 0 ) THEN
91               DO I=1, SIZE
92                  IF ( MPI_REAL .EQ. DATATYPE ) THEN
93                     CALL REAL_RANDOM( Z_COL( I ) )
94                  ELSE
95                     PRINT *, "DATATYPE UNKNOWN"
96                     STOP
97                  ENDIF
98               ENDDO
99               CALL PLA_API_AXPY_VECTOR_TO_GLOBAL_F(
100      &            SIZE, 1.0E00, Z_COL( 1 ), 1, X, 0, IERROR )
101           ENDIF
102   C
103   C Close the objects
104   C
105         CALL PLA_OBJ_API_CLOSE_F( X, IERROR )
106         CALL PLA_OBJ_API_CLOSE_F( A, IERROR )
107   C
108   C Leave Application Interface mode
109   C
110         CALL PLA_API_END_F( IERROR )
111   C
112   C Compute b = A x by performing a matrix-vector multiplication
113   C
114         CALL PLA_CREATE_CONSTANTS_CONF_TO_F( A, MINUS_ONE, ZERO, ONE,
115        &     IERROR )
116         CALL PLA_SYMV_F( PLA_LOWER_TRIANGULAR, ONE, A, X, ZERO, B,
117        &     IERROR )
118   C
119   C Free the temporary objects
120   C
121         CALL PLA_OBJ_FREE_F( MINUS_ONE, IERROR )
122         CALL PLA_OBJ_FREE_F( ZERO, IERROR )
123         CALL PLA_OBJ_FREE_F( ONE, IERROR )
124
125         RETURN
126         END
```

Figure 2.7: A sample subroutine for creating a symmetric positive definite matrix A and vectors x and b s.t. $Ax = b$, Part 3.

```
1          SUBROUTINE PROCESS_ANSWER_F( X )
2
3          IMPLICIT NONE
4
5          INCLUDE "mpif.h"
6          INCLUDE "PLAf.h"
7     C
8     C PLA_OBJ
9     C
10         INTEGER X
11    C
12         INTEGER SIZE, ME, I, IERROR
13    C
14         REAL D_ONE
15         PARAMETER ( D_ONE = 1.0 )
16    C
17         INTEGER MAX_SIZE
18         PARAMETER ( MAX_SIZE = 10000 )
19    C
20         REAL SOLUTION( MAX_SIZE )
21    C
22    C MPI Datatype
23    C
24         INTEGER DATATYPE
25
26         CALL PLA_OBJ_DATATYPE_F( X, DATATYPE, IERROR )
27         CALL PLA_OBJ_GLOBAL_LENGTH_F( X, SIZE, IERROR )
28         IF ( SIZE .GT. MAX_SIZE ) THEN
29            PRINT *, "SIZE too large in process_answer_f"
30            STOP
31         ENDIF
32    C
33    C   What is my rank in the communicator?
34    C
35         CALL MPI_COMM_RANK( MPI_COMM_WORLD, ME, IERROR )
36    C
37    C   Begin Application Interface Mode
38    C
39         CALL PLA_API_BEGIN_F( IERROR )
40    C
41    C     Open object x for read/write mode
42    C
43            CALL PLA_OBJ_API_OPEN_F( X, IERROR )
44
```

Figure 2.8: A sample subroutine for processing the answer of the solve.

```
45              IF ( ME .EQ. 0 ) THEN
46  C
47  C Initialize solution to zero
48  C
49                 IF ( DATATYPE .EQ. MPI_COMPLEX ) THEN
50                    DO I=1, SIZE
51                       SOLUTION( I ) = ( 0.0E00, 0.0E00 )
52                    ENDDO
53                 ELSE IF ( DATATYPE .EQ. MPI_REAL ) THEN
54                    DO I=1, SIZE
55                       SOLUTION( I ) = 0.0E00
56                    ENDDO
57                 ELSE
58                    PRINT *, "Datatype unknown in process_answer"
59                 ENDIF
60  C
61  C Initiate extraction of contents of x to local buffer
62  C
63                 CALL PLA_API_AXPY_GLOBAL_TO_VECTOR_F(
64       &              SIZE, D_ONE, X, 0, SOLUTION, 1, IERROR )
65              ENDIF
66
67  C
68  C Wait until data actually arrives
69  C
70              CALL PLA_OBJ_API_SYNC_F( X, IERROR )
71
72              IF ( ME .EQ. 0 ) THEN
73                 PRINT *, "First computed entry:", SOLUTION( 1 )
74                 PRINT *, "Last  computed entry:", SOLUTION( SIZE )
75              ENDIF
76  C
77  C Close object x
78  C
79           CALL PLA_OBJ_API_CLOSE_F( X, IERROR )
80  C
81  C Exit API mode
82  C
83         CALL PLA_API_END_F( IERROR )
84
85         RETURN
86         END
```

Figure 2.9: A sample subroutine for processing the answer of the solve, Part 2.