2.8 PBLAS routines for matrix-vector operations (Level 2)

Originally, L3 BLAS specification is carried out for the FORTRAN language and the defined/included subroutines are:

a. "General" matrix products (subroutines ending in GEMM):

$$C \leftarrow \alpha\, op(A)\, op(B) + \beta C$$

where op(X) can be X, X^T o X^H

b. Matrix products where one of the matrices is real or symmetrical complex or hermitical

complex (subroutines ending in SYMM or HEMM):

$$C \leftarrow \alpha AB + \beta C \text{ or } C \leftarrow \alpha BA + \beta C$$

where A is symmetrical for SYMM or hermitical for HEMM and is located to the left or right of the multiplication depending on a subroutine parameter (SIDE).

c. Matrices products where one of them is triangular (subroutines ending in TRMM):

$$B \leftarrow \alpha\, op(A)\, B \text{ or } B \leftarrow \alpha\, B\, op(A)$$

where *A* is a triangular matrix; it is to the left or right of the multiplication depending on a subroutine parameter (SIDE), and *op(A)* can be *A, A^T o A^H.*

d. Rank-*k* update of a symmetrical matrix (subroutines ending in SYRK):

$$C \leftarrow \alpha AA^T + \beta C \text{ or } C \leftarrow \alpha A^T A + \beta C$$

where *C* is symmetrical and *A* is to the right or left of the multiplication by A^T, depending on a subroutine parameter (TRANS).

e. Rank-*k* update of a hermitical matrix (subroutines ending in HERK):

$$C \leftarrow \alpha AA^H + \beta C \text{ or } C \leftarrow \alpha A^H A + \beta C$$

where *C* is hermitical and *A* is to the right or left of the multiplication by A^H, depending on a subroutine parameter (TRANS).

f. Rank-2*k* update of a symmetrical matrix (subroutines ending in SYR2K):

$$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C \text{ or } C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$$

where C is symmetrical and A is to the right or left of the multiplication by B^T, depending on a subroutine parameter (TRANS).

g. Rank-2*k* update of a hermitical matrix (subroutines ending in HER2K):

$$C \leftarrow \alpha AB^H + alfa\ BA^H + \beta C \text{ or } C \leftarrow \alpha A^H B + alfa\ B^H A + \beta C$$

where *C* is hermitical and *A* is to the right or left of the multiplication by B^H, depending on a subroutine parameter (TRANS).

h. Solutions to triangular equations systems (subroutines ending in TRSM):

$$B \leftarrow \alpha\, op(A)\, B \text{ or } B \leftarrow \alpha\, B\, op(A)$$

where *A* is a triangular matrix; it is to the left or right of the multiplication depending on a subroutine parameter (SIDE), and *op(A)* can be A^{-1}, A^{-T} *o* A^{-H}.

Matrix multiplication has very specific characteristics as regards the design and implementation of a parallel algorithm in the parallel algorithms context in general:

- Computation independency: each element computed from the result matrix C, c_{ij}, is, in principle, independent of all the other elements. This independence is utterly useful because it allows a wide flexibility degree in terms of parallelization.
- Data independence: the number and type of operations to be carried out

are independent of the data. In this case, the exception is the algorithms of the so-called sparse matrix multiplication, where there exists an attempt to take advantage of the fact that most of the matrices elements to be multiplied (and thus, of the result matrix) are equal to zero.
- Regularity of data organization and of the operations carried out on data: data are organized in two-dimensional structures (the same matrices), and the operations basically consist of multiplication and addition.

Cannon's Algorithm

In computer science, Cannon's algorithm is a distributed algorithm for matrix multiplication for two-dimensional meshes first described in 1969 by Lynn Elliot Cannon. It is also proposed for a two-dimensional array of processing elements interconnected as a mesh and with the edges of each row and column interconnected, i.e. making up a structure called torus for 3×3 processing elements.

Initially, matrices *A, B,* and *C* data distribution is similar to that defined previously in the mesh, i.e. if the processors are numbered according to their position in the two-dimensional array, processor P_{ij} $(0 \leq i, j \leq P-1)$ has the elements or blocks of position ij $(0 \leq i, j \leq P-1)$ of matrices *A, B* and *C*. In order to simplify the explanation, matrices elements shall be used instead of blocks. From this data distribution, matrices *A* and *B* data are "realigned" or reassigned so that, if there is a two-dimensional array of $P \times P$ processors, the element or submatrix *A* in row *i* and column $(j+i) \bmod P$, $a_{i,(j+i) \bmod P}$, and also the element or submatrix of *B* in row $(i+j) \bmod P$ and column *j*, $b_{(i+j) \bmod P,i}$, are assigned to processor P_{ij}. In other words, each data of row i $(0 \leq i \leq P-1)$ of the elements or submatrices of *A* are transferred or shifted *i* times towards the left processors, and each data or column j $(0 \leq j \leq P-1)$ of the elements or submatrices of *B* are transferred or shifted j times towards upper processors.

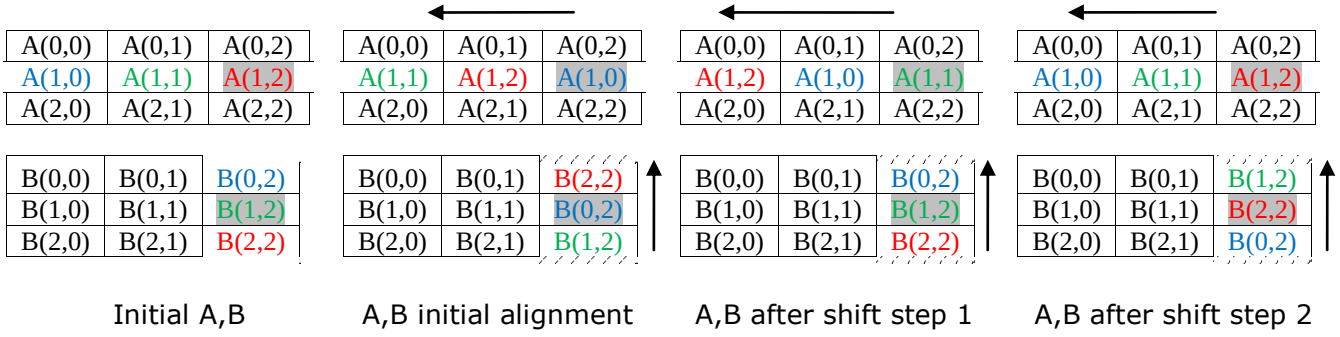
From the initial relocation, the following steps are carried out iteratively:
- Local multiplication of data assigned in each processor for a partial result computation;
- Left rotation of the elements or submatrices of *A*;
- Upwards rotation of the elements or submatrices of *B*, and after *P* of these steps, thoroughly computed values of matrix *C* are finally obtained.

Summarizing, the outstanding characteristics of this way to carry out matrix multiplication are:
- By the way matrices *A* and *B* data are communicated, this is an "initial alignment and rotating" algorithm.
- Load balance, both in terms of computation and communications, is assured only if the processing algorithms are homogeneous.
- As in the processing defined for the processors grid, the running time is minimized if the computation can be overlapped in time with communications.
- Matrices *A* and *B* data distribution is not the initial one when the matrix multiplication computation is finished.

Finally, the **Cannon's Algorithm** consists of the following steps:

1) The **initial step** of the algorithm regards the alignment of the matrixes:
   a. Align the blocks of A and B in such a way that each process can independently start multiplying its local submatrices. This is done by shifting all submatrices $A_{i,j}$ to the left (with wraparound) by **i** steps a and all submatrices $B_{i,j}$ up (with wraparound) by **j** steps.
2) Perform local block multiplication.
3) Each block of *A* moves **one step left** and each block of *B* moves **one step up** (again with wraparound).
4) Perform next block multiplication, add to partial result, repeat until all blocks have been multiplied.

Cannon's Algorithm for 3×3 Matrices and process (1,2) to determine C_{12}



So C(1,2)=A(1,0)*B(0,2)+A(1,1)*B(1,2)+ A(1,2)*B(2,2) and according the Cannon's Algorithm we have:

- $C^1(1,2)= A(1,0)*B(0,2)$ (after A, B initial alignment)
- $C^2(1,2)= C^1(1,2)+A(1,1)*B(1,2)$ (A, B after shift step 1)
- $C(1,2)=C^3(1,2)= C^2(1,2)+A(1,2)*B(2,2)$ (A, B after shift step 2)

Attention! Limitations of Cannon's Algorithm

- P (number of processors) must be a perfect square;
- Matrices A and B must be square, and evenly divisible by √p

```
PvGEMV(TRANS,M,N,ALPHA,A,IA,JA,DESCA,X,IX,JX,DESCX,INCX,
       BETA,Y,IY,JY,DESCY,INCY)
void pdgemv_(char *trans, int *m, int *n, double *alpha,
       double *A, int *ia, int *ja, int *desc_A, double *X,
       int *ix, int *jx,  int*desc_X, int *incx, double *beta,
       double *Y, int *iy, int *jy, int *desc_Y, int *incy );
```

Purpose PvGEMV performs one of the distributed matrix-vector operations
*sub(Y) := alpha*sub(A)×sub(X) + beta×sub(Y)*, or
sub(Y) := alpha×sub(A)^T × sub(X) + beta×sub(Y),
where sub(A) denotes A(IA:IA+M-1,JA:JA+N-1),

$$\text{sub}(X) = \begin{cases} \begin{cases} X(IX:IX,\ JX:JX+N-1) & \text{if } INCX = M\_X \\ X(IX:IX+N-1,\ JX:JX) & \text{if } INCX = 1 \text{ and } INCX <> M\_X \end{cases} & \text{if } TRANS = 'N' \\ \begin{cases} X(IX:IX,\ JX:JX+M-1) & \text{if } INCX = M\_X \\ X(IX:IX+M-1,\ JX:JX) & \text{if } INCX = 1 \text{ and } INCX <> M\_X \end{cases} & \text{else} \end{cases}$$

$$\text{sub}(Y) = \begin{cases} \begin{cases} Y(IY:IY,\ JY:JY+M-1) & \text{if } INCY = M\_Y \\ Y(IY:IY+M-1,\ JY:JY) & \text{if } INCY = 1 \text{ and } INCY <> M\_X \end{cases} & \text{if } TRANS = 'N' \\ \begin{cases} Y(IY:IY,\ JY:JY+N-1) & \text{if } INCY = M\_Y \\ Y(IY:IY+N-1,\ JY:JY) & \text{if } INCY = 1 \text{ and } INCXY <> M\_Y \end{cases} & \text{else} \end{cases}$$

alpha and *beta* are scalars, and *sub(X)* and *sub(Y)* are distributed vectors and *sub(A)* is a *M*-by-*N* distributed submatrix.

Arguments

TRANS (global input) CHARACTER

On entry, TRANS specifies the operation to be performed as follows:

$$\begin{cases} \text{sub}(Y) := \text{alpha} * \text{sub}(A) * \text{sub}(X) + \text{beta} * \text{sub}(Y), & \text{if } TRANS = 'N' \\ \text{sub}(Y) := \text{alpha} * \text{sub}(A)' * \text{sub}(X) + \text{beta} * \text{sub}(Y), & \text{if } TRANS = 'T' \\ \text{sub}(Y) := \text{alpha} * \text{sub}(A)' * \text{sub}(X) + \text{beta} * \text{sub}(Y). & \text{if } TRANS = 'C' \end{cases}$$

M (global input) INTEGER

The number of rows to be operated on i.e. the number of rows of the distributed submatrix sub(A). M≥ 0.

N (global input) INTEGER

The number of columns to be operated on i.e the number of columns of the distributed submatrix sub(A). N≥ 0.

ALPHA (global input) REAL/COMPLEX

On entry, ALPHA specifies the scalar alpha.

A (local input) array of dimension (LLD_A,LOCq(JA+N-1))

This array contains the local pieces of the distributed matrix sub(A).

IA (global input) INTEGER

The global row index of the submatrix of the distributed matrix A to operate on.

JA (global input) INTEGER

The global column index of the submatrix of the distributed matrix A to operate on.

DESCA (global and local input) INTEGER array of dimension 8

The array descriptor of the distributed matrix A.

X (local input/local output) array of dimension at least

$$\begin{cases} ((JX-1) * M\_X + IX + (N-1) * \text{abs}(INCX)) & \text{if } TRANS = 'N' \\ ((JX-1) * M\_X + IX + (M-1) * \text{abs}(INCX)) & \text{else.} \end{cases}$$

This array contains the entries of the distributed vector sub(X).

IX (global input) INTEGER

The global row index of the submatrix of the distributed matrix X to operate on.

JX (global input) INTEGER

The global column index of the submatrix of the distributed matrix X to

operate on.

DESCX (global and local input) INTEGER array of dimension 8

The array descriptor of the distributed matrix X.

INCX (global input) INTEGER

The global increment for the elements of X. Only two values of INCX are supported in this version, namely 1 and M_X.

BETA (global input) REAL/COMPLEX

On entry, BETA specifies the scalar beta. When BETA is supplied as zero then sub(Y) need not be set on input.

Y (local input/local output) array of dimension at least

$$\begin{Bmatrix} ((JY - 1) * M\_Y + IY + (M - 1) * \text{abs}(INCY)) & \text{if TRANS} = \text{'N'} \\ ((JY - 1) * M\_Y + IY + (N - 1) * \text{abs}(INCY)) & \text{else} \end{Bmatrix}$$

This array contains the entries of the distributed vector sub(Y). On exit, sub(Y) is overwritten by the updated distributed vector sub(Y).

IY (global input) INTEGER

The global row index of the submatrix of the distributed matrix Y to operate on.

JY (global input) INTEGER

The global column index of the submatrix of the distributed matrix Y to operate on.

DESCY (global and local input) INTEGER array of dimension 8

The array descriptor of the distributed matrix Y.

INCY (global input) INTEGER

The global increment for the elements of Y. Only two values of INCY are supported in this version, namely 1 and M_Y.

Notes and Coding Rules

- These subroutines accept lowercase letters for the transa argument.
- For PDGEMV, if you specify 'C' for transa, it is interpreted as though you specified 'T'.
- The matrix and vectors must have no common elements; otherwise, results are unpredictable.
- The NUMROC utility subroutine can be used to determine the values of LOCp(M_) and LOCq(N_) used in the argument descriptions above.
- The following values must be equal: CTXT_A = CTXT_X = CTXT_Y.
- The following coding rules depend upon the values specified for transa and incx:
  1. If transa = 'N' and incx = M_X:
     1) The following block sizes must be equal: NB_A = NB_X.
     2) In the process grid, the process column containing the first column of the submatrix x must also contain the first column of the submatrix A; that is, iacol = ixcol, where:
        1. iacol = mod((((ja-1)/NB_A)+CSRC_A), q)
        2. ixcol = mod((((jx-1)/NB_X)+CSRC_X), q)
     3) The block column offset of x must be equal to the block column offset of A; that is, mod(jx-1, NB_X) = mod(ja-1, NB_A).
  2. If transa = 'N' and incx = 1(≠ M_X):
     1) The following block sizes must be equal: NB_A = MB_X.
     2) The block row offset of x must be equal to the block column

offset of *A*; that is, mod(ix-1, MB_X) = mod(ja-1, NB_A).

3. If transa = 'T' or 'C' and incx = M_X:
    1) The following block sizes must be equal: MB_A = NB_X.
    2) The block column offset of *x* must be equal to the block row offset of *A*; that is, mod(jx-1, NB_X) = mod(ia-1, MB_A).
4. If transa = 'T' or 'C' and incx = 1(≠ M_X):
    1) The following block sizes must be equal: MB_A = MB_X.
    2) In the process grid, the process row containing the first row of the submatrix *X* must also contain the first row of the submatrix *A*; that is, iarow = ixrow, where:
        1. iarow = mod((((ia-1)/MB_A)+RSRC_A), p)
        2. ixrow = mod((((ix-1)/MB_X)+RSRC_X), p)
    3) The block row offset of *x* must be equal to the block row offset of *A*; that is, mod(ix-1, MB_X) = mod(ia-1, MB_A).

- The following coding rules depend upon the values specified for transa and incy:
    1. If transa = 'N' and incy = M_Y:
        1) The following block sizes must be equal: MB_A = NB_Y.
        2) The block column offset of *y* must be equal to the block row offset of *A*; that is, mod(jy-1, NB_Y) = mod(ia-1, MB_A).
    2. If transa = 'N' and incy = 1(≠ M_Y):
        1) The following block sizes must be equal: MB_A = MB_Y.
        2) In the process grid, the process row containing the first row of the submatrix *Y* must also contain the first row of the submatrix *A*; that is, iarow = iyrow, where:
            1. iarow = mod((((ia-1)/MB_A)+RSRC_A), p)
            2. iyrow = mod((((iy-1)/MB_Y)+RSRC_Y), p)
        3) The block row offset of *y* must be equal to the block row offset of *A*; that is, mod(iy-1, MB_Y) = mod(ia-1, MB_A).
    3. If transa = 'T' or 'C' and incy = M_Y:
        1) The following block sizes must be equal: NB_A = NB_Y.
        2) In the process grid, the process column containing the first column of the submatrix *Y* must also contain the first column of the submatrix *A*; that is, iacol = iycol, where:
            1. iacol = mod((((ja-1)/NB_A)+CSRC_A), q)
            2. iycol = mod((((jy-1)/NB_Y)+CSRC_Y), q)
        3) The block column offset of y must be equal to the block column offset of A; that is, mod(jy-1, NB_Y) = mod(ja-1, NB_A).
    4. If transa = 'T' or 'C' and incy = 1(≠ M_Y):
        1) The following block sizes must be equal: NB_A = MB_Y.
        2) The block row offset of *y* must be equal to the block column offset of *A*; that is, mod(iy-1, MB_Y) = mod(ja-1, NB_A).

```
PvTRMV(UPLO,TRANS,DIAG,N,A,IA,JA,DESCA,X,IX,JX,DESCX,INCX)
```

Purpose: PvTRMV performs one of the distributed matrix-vector operations

sub(X) := sub(A) × sub(X) or sub(X) := sub(A)^T ×sub(X),
where sub(A) denotes A(IA:IA+N-1,JA:JA+N-1),

$$sub(X) = \left\{ \begin{array}{l} X(\ IX,\ JX\ :\ JX\ +\ N\ -1\ ) \quad \text{if INCX = M\_X,} \\ \\ X(\ IX\ :\ IX\ +\ N\ -1,\ JX\ ) \quad \text{if INCX = 1 and INCX <> M\_X,} \end{array} \right\}$$

sub(X) is an *N* element vector and *sub(A)* is an *N*-by-*N* unit, or non-unit, upper or lower triangular distributed matrix.

Arguments

UPLO (global input) CHARACTER

On entry, UPLO specifies whether the upper or lower triangular part of the distributed matrix sub(A) is to be referenced.

TRANS (global input) CHARACTER

On entry, TRANS specifies the operation to be performed as follows:
*TRANS = 'N' sub(x) := sub(A)*sub(x).*
*TRANS = 'T' sub(x) := sub(A)'*sub(x).*
*TRANS = 'C' sub(x) := sub(A)'*sub(x).*

DIAG (global input) CHARACTER

On entry, DIAG specifies whether or not sub(A) is unit triangular.

N (global input) INTEGER

The order of the distributed matrix sub(A). N≥0.

A (local input) array of dimension (LLD_A,LOC_q(JA+N-1))

This array contains the local pieces of the distributed matrix sub(A). Before entry with UPLO = 'U', the leading N-by-N upper triangular part of the distributed matrix sub(A) must contain the upper triangular distributed matrix and the strictly lower triangular part of sub(A) is not referenced. Before entry with UPLO = 'L', the leading N-by-N lower triangular part of the distributed matrix sub(A) must contain the lower triangular distributed matrix and the strictly upper triangular part of sub(A) is not referenced. Note that when DIAG = 'U', the diagonal elements of sub(A) are not referenced either, but are assumed to be unity.

IA (global input) INTEGER

The global row index of the submatrix of the distributed matrix A to operate on.

JA (global input) INTEGER

The global column index of the submatrix of the distributed matrix A to operate on.

DESCA (global and local input) INTEGER array of dimension 8

The array descriptor of the distributed matrix A.

X (local input/local output) array of dimension at least ((JX-1)*M_X+IX+(N-1)*abs(INCX))

This array contains the entries of the distributed vector sub(X).

IX (global input) INTEGER

The global row index of the submatrix of the distributed matrix X to operate on.

JX (global input) INTEGER

The global column index of the submatrix of the distributed matrix X to operate on.

DESCX (global and local input) INTEGER array of dimension 8

The array descriptor of the distributed matrix X.

INCX (global input) INTEGER

The global increment for the elements of X. Only two values of INCX are supported in this version, namely 1 and M_X.

```
PvTRSV(UPLO,TRANS,DIAG,N,A,IA,JA,DESCA,X,IX,JX,DESCX,INCX)
void pdtrsv_(F_CHAR_T UPLO, F_CHAR_T TRANS, F_CHAR_T DIAG,
           int *N, double *A, int *IA, int *JA, int *DESCA,
           double *X, int *IX, int *JX, int *DESCX, int *INCX)
```

Purpose: PvTRSV solves one of the systems of equations

sub(A) × sub(X) = b, or sub(A)^T× sub(X) = b,

where sub(A) denotes A(IA:IA+N-1,JA:JA+N-1),

$$sub(X) = \left\{ \begin{array}{ll} X(\ IX,\ JX\ :\ JX\ +\ N\ -1\ ) & \text{if INCX = M\_X,} \\ X(\ IX\ :\ IX\ +\ N\ -1,\ JX\ ) & \text{if INCX = 1 and INCX <> M\_X,} \end{array} \right\}$$

b and *sub(X)* are *N* element distributed vectors and *sub(A)* is an *N*-by-*N* unit, or non-unit, upper or lower triangular distributed matrix. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine. The term *b* used in the systems of equations listed above represents the right-hand side of the system. It is important to note that in these subroutines the right-hand side of the equation is actually provided in the input-output argument x.

Arguments

UPLO (global input) CHARACTER

On entry, UPLO specifies whether the distributed matrix sub(A) is an upper or lower triangular matrix.

TRANS (global input) CHARACTER

On entry, TRANS specifies the operation to be performed as follows:

TRANS = 'N' sub(A) × sub(X) = b.

TRANS = 'T' sub(A)^T× sub(X) = b.

TRANS = 'C' sub(A)^T× sub(X) = b.

DIAG (global input) CHARACTER

On entry, DIAG specifies whether or not sub(A) is unit triangular as follows:

DIAG = 'U' sub(A) is assumed to be unit triangular.

DIAG = 'N' sub(A) is not assumed to be unit triangular.

N (global input) INTEGER

The order of the distributed matrix sub(A). N≥0. A (local input) array of dimension (LLD_A, LOCq(JA+N-1)).This array contains the local pieces of the distributed matrix sub(A). Before entry with UPLO = 'U', the leading N-by-N upper triangular part of the distributed matrix sub(A) must contain the upper triangular distributed matrix and the strictly lower triangular part of sub(A) is not referenced. Before entry with UPLO = 'L', the leading N-by-N lower triangular part of the distributed matrix sub(A) must contain the lower triangular distributed matrix and the strictly upper triangular part of sub(A) is not referenced. Note that when DIAG = 'U', the diagonal elements of sub(A) are not referenced either, but are assumed to be unity.

IA (global input) INTEGER

The global row index of the submatrix of the distributed matrix A to operate on.

JA (global input) INTEGER

The global column index of the submatrix of the distributed matrix A to operate on.

DESCA (global and local input) INTEGER array of dimension 8

The array descriptor of the distributed matrix A.

X (local input/local output) array of dimension at least ((JX-

1)*M_X+IX+(N-1)*abs(INCX))

This array contains the entries of the distributed vector sub(X). Before entry, sub(X) must contain the N element right-hand side distributed vector b. On exit, sub(X) is overwritten with the solution vector.

IX (global input) INTEGER

The global row index of the submatrix of the distributed matrix X to operate on.

JX (global input) INTEGER

The global column index of the submatrix of the distributed matrix X to operate on.

DESCX (global and local input) INTEGER array of dimension 8

The array descriptor of the distributed matrix X.

INCX (global input) INTEGER

The global increment for the elements of X. Only two values of INCX are supported in this version, namely 1 and M_X.

Notes and Coding Rules

1) These subroutines accept lowercase letters for the uplo, transa, and diag arguments.
2) For PDTRSV, if you specify 'C' for transa, it is interpreted as though you specified 'T'.
3) The matrix and vector must have no common elements; otherwise, results are unpredictable.
4) PDTRSV and PZTRSV assume certain values in your array for parts of a triangular matrix. For unit triangular matrices, the elements of the diagonal are assumed to be one. When using an upper or lower triangular matrix, the unreferenced elements in the strictly lower or upper triangular part, respectively, are assumed to be zero. As a result, you do not have to set these values.
5) The NUMROC utility subroutine can be used to determine the values of LOCp(M_) and LOCq(N_) used in the argument descriptions above.
6) The following values must be equal: CTXT_A = CTXT_X.
7) The global triangular matrix *A* must be distributed using a square block-cyclic distribution; that is, MB_A = NB_A.
8) The block row and block column offsets of the global triangular matrix *A* must be equal; that is, mod(ia-1,MB_A) = mod(ja-1,NB_A).
9) If incx = M_X:
    a. The following block sizes must be equal: NB_X = MB_A = NB_A
    b. If transa = 'T', then (in the process grid) the process column containing the first column of the submatrix *A* must also contain the first column of the submatrix *X*; that is, iacol = ixcol, where:
        1. iacol = mod((((ja-1)/NB_A)+CSRC_A), q)
        2. ixcol = mod((((jx-1)/NB_X)+CSRC_X), q)
    c. The block column offset of *x* must be equal to the block row and block column offsets of *A*; that is, mod(jx-1, NB_X) = mod(ja-1, NB_A) = mod(ia-1, MB_A).
10) If incx = 1(≠ M_X):
    a. The following block sizes must be equal: MB_X = MB_A = NB_A
    b. If transa = 'N', then (in the process grid) the process row

containing the first row of the submatrix *A* must also contain the first row of the submatrix *X*; that is, iarow = ixrow, where:

1. iarow = mod((((ia-1)/MB_A)+RSRC_A), p)
2. ixrow = mod((((ix-1)/MB_X)+RSRC_X), p)

c. The block row offset of *x* must be equal to the block row and block column offsets of *A*; that is, mod(ix-1, MB_X) = mod(ia-1, MB_A) = mod(ja-1, NB_A).

Example 2.8.1*. The using of functions pdgemv_ and pdtrsv_*

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "mpi.h"
#include <iostream>
#include <iomanip>
#include <fstream>
#include <sstream>
using namespace std;
#define AA(i,j) AA[(i)*M+(j)]
#define AATRI(i,j) AATRI[(i)*M+(j)]
extern "C"
{
void Cblacs_pinfo( int* mypnum, int* nprocs);
void Cblacs_get( int context, int request, int* value);
int  Cblacs_gridinit( int* context, char * order, int np_row, int np_col);
void Cblacs_gridinfo( int context, int*  np_row, int* np_col, int*  my_row,
int*  my_col);
void Cblacs_gridexit( int context);
void Cblacs_barrier(int, const char*);
void Cblacs_exit( int error_code);
void Cblacs_pcoord(int, int, int*, int*);
int  numroc_( int *n, int *nb, int *iproc, int *isrcproc, int *nprocs);
int indxl2g_(int*, int*, int*, int*, int*);
void descinit_( int *desc, int *m, int *n, int *mb, int *nb, int *irsrc, int
*icsrc, int *ictxt, int *lld, int *info);
void pdgemv_(char *trans, int *m, int *n, double *alpha, double *A, int *ia,
int *ja, int *desc_A, double *X, int *ix,
             int *jx, int*desc_X, int *incx, double *beta, double *Y, int
*iy, int *jy, int *desc_Y, int *incy );
void pdtrsv_(char *UPLO, char *trans, char *DIAG, int *N, double *A, int *IA,
int *JA, int *DESCA, double *X, int *IX,
              int *JX, int *DESCX, int *INCX);
} // extern "C"
int main(int argc, char **argv) {
   int i, j, k;
/************  MPI ***************************/
   int myrank_mpi, nprocs_mpi;
   MPI_Init( &argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank_mpi);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs_mpi);
/************  BLACS ***************************/
   int ictxt, nprow, npcol, myrow, mycol,nb,myid;
   int info,itemp;
   int ZERO=0,ONE=1;
   nprow = 2; npcol = 2; nb =2;
Cblacs_pinfo( &myrank_mpi, &nprocs_mpi) ;
```

```
Cblacs_get( -1, 0, &ictxt );
Cblacs_gridinit( &ictxt, "Row", nprow, npcol );
Cblacs_gridinfo( ictxt, &nprow, &npcol, &myrow, &mycol );
   int M=5;
   double *AA = (double*) malloc(M*M*sizeof(double));// matricea "globala"
pentru inmultire matrice cu vectorAA*X=Y
   double *AATRI = (double*) malloc(M*M*sizeof(double));//matricea "globala"
rezolv. sitemului de ecuatii AATRI*Y=B
   for(i=0;i<M;i++ )
     for(j=0;j<M;j++)
        AA[i*M+j]=(2*i+3*j+1);
//Se complecteaza cu valori matricea "triungiulara"
for(i=0;i<M;i++ )
     for(j=0;j<M;j++)
{if (j>i) AATRI[i*M+j]=0;
 else       AATRI[i*M+j]=(2*i+3*j+1);
}
// Tipar Matricele globale (initiale)
if (myrank_mpi==0)
{
printf("============ REZULT OF THE PROGRAM %s \n",argv[0]);
cout << "Global matrix AA:\n";
              for (i = 0; i < M; ++i) {
                  for (int j = 0; j < M; ++j) {
                      cout << setw(3) << *(AA + M*i + j) << " ";
                  }
                  cout << "\n";
              }
              cout << endl;
cout << "Global matrix AATRI:\n";
              for (i = 0; i < M; ++i) {
                  for (int j = 0; j < M; ++j) {
                      cout << setw(3) << *(AATRI + M*i + j) << " ";
                  }
                  cout << "\n";
              }
              cout << endl;
}
   double *X = (double*) malloc(M*sizeof(double)); // X- vectorul "global"
   double *XTRI = (double*) malloc(M*sizeof(double)); // XTRI- vectorul
"global"(partea dreapta a siste. de ecuatii)
   X[0]=1;X[1]=1;X[2]=0;X[3]=0;X[4]=1;
   XTRI[0]=1;XTRI[1]=1;XTRI[2]=0;XTRI[3]=0;XTRI[4]=1;
   int descA[9],descAtri[9],descx[9],descxtri[9],descy[9];
  int mA = numroc_( &M, &nb, &myrow, &ZERO, &nprow );
   int nA = numroc_( &M, &nb, &mycol, &ZERO, &npcol );
   int mAtri = numroc_( &M, &nb, &myrow, &ZERO, &nprow );
   int nAtri = numroc_( &M, &nb, &mycol, &ZERO, &npcol );
   int nx = numroc_( &M, &nb, &myrow, &ZERO, &nprow );
   int my = numroc_( &M, &nb, &myrow, &ZERO, &nprow );
   descinit_(descA, &M,   &M,   &nb,  &nb,  &ZERO, &ZERO, &ictxt, &mA,
&info);
   descinit_(descAtri, &M,   &M,   &nb,  &nb,  &ZERO, &ZERO, &ictxt, &mA,
&info);
   descinit_(descx, &M, &ONE,   &nb, &ONE,  &ZERO, &ZERO, &ictxt, &nx,
&info);
   descinit_(descxtri, &M, &ONE,   &nb, &ONE,  &ZERO, &ZERO, &ictxt, &nx,
```

```
&info);
   descinit_(descy, &M, &ONE,   &nb, &ONE,  &ZERO, &ZERO, &ictxt, &my,
&info);
   double *x = (double*) malloc(nx*sizeof(double)); //x- vectorul local
   double *xtri = (double*) malloc(nx*sizeof(double)); //xtri- vectorul local
   double *y = (double*) calloc(my,sizeof(double));
   double *A = (double*) malloc(mA*nA*sizeof(double)); //matricea locala p-ru
inlultirea cu vector
   double *Atri = (double*) malloc(mA*nA*sizeof(double)); //matricea locala
p-ru rez. sist. de ecuatii
   int sat,sut;
// Se complecteaza cu valor vectorul si matricea locala
for(i=0;i<mA;i++)
   for(j=0;j<nA;j++){
                sat= (myrow*nb)+i+(i/nb)*nb;
                sut= (mycol*nb)+j+(j/nb)*nb;
                A[j*mA+i]=AA(sat,sut);
                Atri[j*mA+i]=AATRI(sat,sut);
        }
   for(i=0;i<nx;i++){
                sut= (myrow*nb)+i+(i/nb)*nb;
                x[i]=X[sut];
            xtri[i]=X[sut];
        }
   double alpha = 1.0; double beta = 0.0;
pdgemv_("N",&M,&M,&alpha,A,&ONE,&ONE,descA,x,&ONE,&ONE,descx,&ONE,&beta,y,&ON
E,&ONE,descy,&ONE);
if (myrank_mpi==0) printf("Final rezult of matrix-vector multiplication \n");
if (mycol==0)
{
 for(i=0;i<my;i++)
   printf("For rank=%d y[%d](gl.ind=%d) =%.2f \n", myrank_mpi,i,indxl2g_(&i,
&nb, &myrow, &ZERO, &nprow),y[i]);
}
Cblacs_barrier(ictxt, "All");
pdtrsv_("L", "N", "N", &M, Atri, &ONE, &ONE, descAtri, xtri,
&ONE,&ONE,descxtri, &ONE);
if (myrank_mpi==0) printf("Final rezult of solves of the systems of equations
\n");
Cblacs_barrier(ictxt, "All");
if (mycol==0)
{
for(i=0;i<nx;i++)
printf("For rank=%d x[%d](gl.ind=%d) =%.2f \n", myrank_mpi,i,indxl2g_(&i,
&nb, &myrow, &ZERO, &nprow),xtri[i]);
}
Cblacs_barrier(ictxt, "All");
   Cblacs_gridexit(0);
   MPI_Finalize();
   return 0;
}
```

Result of compilation and executing

```
[MI_gr_TPS1@hpc]$ ./mpiCC_ScL -o Example2.8.1.exe Example2.8.1.cpp
[MI_gr_TPS1@hpc]$ opt/openmpi/bin/mpirun -n 4 -host compute-0-0
```

```
Example2.8.1.cpp

Global matrix AA:
  1   4   7  10  13
  3   6   9  12  15
  5   8  11  14  17
  7  10  13  16  19
  9  12  15  18  21
Global matrix AATRI:
  1   0   0   0   0
  3   6   0   0   0
  5   8  11   0   0
  7  10  13  16   0
  9  12  15  18  21
Final rezult of matrix-vector multiplication
For rank=0 y[0](gl.ind=0) =18.00
For rank=0 y[1](gl.ind=1) =24.00
For rank=2 y[0](gl.ind=2) =30.00
For rank=2 y[1](gl.ind=3) =36.00
For rank=0 y[2](gl.ind=2) =42.00
Final rezult of solves of the systems of equations
For rank=0 x[0](gl.ind=0) =1.00
For rank=0 x[1](gl.ind=1) =-0.33
For rank=2 x[0](gl.ind=2) =-0.21
For rank=2 x[1](gl.ind=3) =-0.06
For rank=0 x[2](gl.ind=2) =0.01
```

Additional explanations for Example 2.8.1.

Let $\mathbf{A}=\begin{pmatrix}1&4&7&10&13\\3&6&9&12&15\\5&8&11&14&17\\7&10&13&16&19\\9&12&15&18&21\end{pmatrix}$ and $\mathbf{X}=\begin{pmatrix}1\\1\\0\\0\\1\end{pmatrix}$. It is evident that $\mathbf{AX}=\begin{pmatrix}18\\24\\30\\36\\42\end{pmatrix}$

If the matrix **A** is distributed to the grid with **2x2** processes with block dimensions **2x2** using *2-D cyclic* algorithm, then the processes will possess the following submatrices:

$\mathbf{A_{(0,0)}}=\begin{pmatrix}1&4&13\\3&6&15\\9&12&21\end{pmatrix}$, $\mathbf{X_{(0,0)}}=\begin{pmatrix}1\\1\\1\end{pmatrix}$; $\mathbf{A_{(0,1)}}=\begin{pmatrix}7&10\\9&12\\15&18\end{pmatrix}$, $\mathbf{X_{(0,1)}}=\begin{pmatrix}1\\1\\1\end{pmatrix}$;

$\mathbf{A_{(1,0)}}=\begin{pmatrix}5&8&17\\7&10&19\end{pmatrix}$, $\mathbf{X_{(1,0)}}=\begin{pmatrix}0\\0\end{pmatrix}$; $\mathbf{A_{(1,1)}}=\begin{pmatrix}11&14\\13&16\end{pmatrix}$, $\mathbf{X_{(1,1)}}=\begin{pmatrix}0\\0\end{pmatrix}$;

So we obtain:

	0	**1**
0	$A_{(0,0)}$	$A_{(0,1)}$
1	$A_{(1,0)}$	$A_{(1,1)}$

	0
0	$X_{(0,0)}$
1	$X_{(1,0)}$

The process will be carried out the following calculations:

- Process(0,0): $Y_{(0,0)}= A_{(0,0)}\ X_{(0,0)}+ A_{(0,1)}\ X_{(1,0)}$
- Process(1,0): $Y_{(1,0)}= A_{(1,0)}\ X_{(0,0)}+ A_{(1,1)}\ X_{(1,0)}$

To determine the product **AX** we use the Cannon algorithm**.** Therefore:

1. It is carried out the alignment of submatrices, i.e. the submatrices $\boldsymbol{A_{(i,j)}}$ (that are in possession of the process *(i,j))* are "moving" (are transmitted on process grid) to the left by *i* times and the submatrices $\boldsymbol{X_{(i,j)}}$ are "moving"" (are transmitted on process grid) to the top by *j* times.

   As a result for each process we get the following:

   - The process (0,0) does nothing (because it can perform the operation of matrix- vector multiplication) and thus it is realized the product $\boldsymbol{Y^0_{(0,0)} = A_{(0,0)}X_{(0,0)}} = \begin{pmatrix} 1 & 4 & 13 \\ 3 & 6 & 15 \\ 9 & 12 & 21 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 18 \\ 24 \\ 42 \end{pmatrix}$

- The process (1,0). The submatrix $\boldsymbol{A_{(1,0)}}$ is transmitted by $i=1$ to the left, but $\boldsymbol{X_{(1,0)}}$ is not transmitted ($j=0$), so we have:

	0	1
0	$A_{(0,0)}$	$A_{(0,1)}$
1	$A_{(1,1)}$	$A_{(1,0)}$

	0
0	$X_{(0,0)}$
1	$X_{(1,0)}$

So the process (1,0) can calculate $\boldsymbol{Y^0_{(1,0)} = A_{(1,1)}\, X_{(1,0)}} = \begin{pmatrix} 11 & 14 \\ 13 & 16 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$.

2. Each submatrices of $\boldsymbol{A_{(i,j)}}$ (that are in possession of the process *(i,j) after the iteration 1.)* moves **one step left** and each submatrices $\boldsymbol{X_{(i,j)}}$ moves **one step up** (again with wraparound).

   As a result for each process we get the following:

   - The process (0,0):

	0	1
0	$A_{(0,1)}$	$A_{(0,0)}$
1	$A_{(1,0)}$	$A_{(1,1)}$

	0
0	$X_{(1,0)}$
1	$X_{(0,0)}$

   The process will be carried out the following calculations $\boldsymbol{Y^1_{(0,0)} = Y^0_{(0,0)} + A_{(0,1)}\, X_{(1,0)}} = \begin{pmatrix} 18 \\ 24 \\ 42 \end{pmatrix} + \begin{pmatrix} 7 & 10 \\ 9 & 12 \\ 15 & 18 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 18 \\ 24 \\ 42 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$

- The process (1,0):

	0	1
0	$A_{(0,0)}$	$A_{(0,1)}$
1	$A_{(1,0)}$	$A_{(1,1)}$

	0
0	$X_{(1,0)}$
1	$X_{(0,0)}$

   The process will be carried out the following calculations $\boldsymbol{Y^1_{(1,0)} = Y^0_{(1,0)} + A_{(1,0)}\, X_{(0,0)}} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 5 & 8 & 17 \\ 7 & 10 & 19 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 30 \\ 36 \end{pmatrix}$

Therefore, finally we obtain

$$Y=AX=\begin{pmatrix} 18 \\ 24 \\ 30 \\ 36 \\ 42 \end{pmatrix} \begin{matrix} \rightarrow process \quad (0,0) \\ \rightarrow process \quad (0,0) \\ \rightarrow process \quad (1,0) \\ \rightarrow process \quad (1,0) \\ \rightarrow process \quad (0,0) \end{matrix}$$

PBLAS Level 2 Routine Groups and Their Data Types

Routine Groups	Data Types	Description
p?gemv	s, d, c, z	Matrix-vector product using a distributed general matrix
p?agemv	s, d, c, z	Matrix-vector product using absolute values for a distributed general matrix
p?ger	s, d	Rank-1 update of a distributed general matrix
p?gerc	c, z	Rank-1 update (conjugated) of a distributed general matrix
p?geru	c, z	Rank-1 update (unconjugated) of a distributed general matrix
p?hemv	c, z	Matrix-vector product using a distributed Hermitian matrix
p?ahemv	c, z	Matrix-vector product using absolute values for a distributed Hermitian matrix
p?her	c, z	Rank-1 update of a distributed Hermitian matrix
p?her2	c, z	Rank-2 update of a distributed Hermitian matrix
p?symv	s, d	Matrix-vector product using a distributed symmetric matrix
p?asymv	s, d	Matrix-vector product using absolute values for a distributed symmetric matrix
p?syr	s, d	Rank-1 update of a distributed symmetric matrix
p?syr2	s, d	Rank-2 update of a distributed symmetric matrix
p?trmv	s, d, c, z	Distributed matrix-vector product using a triangular matrix
p?atrmv	s, d, c, z	Distributed matrix-vector product using absolute values for a triangular matrix
p?trsv	s, d, c, z	Solves a system of linear equations whose coefficients

Routine Groups	Data Types	Description
		are in a distributed triangular matrix

Routine Group	Data Types	Description
p?geadd	s, d, c, z	Distributed matrix-matrix sum of general matrices
p?tradd	s, d, c, z	Distributed matrix-matrix sum of triangular matrices
p?gemm	s, d, c, z	Distributed matrix-matrix product of general matrices
p?hemm	c, z	Distributed matrix-matrix product, one matrix is Hermitian
p?herk	c, z	Rank-k update of a distributed Hermitian matrix
p?her2k	c, z	Rank-2k update of a distributed Hermitian matrix
p?symm	s, d, c, z	Matrix-matrix product of distributed symmetric matrices
p?syrk	s, d, c, z	Rank-k update of a distributed symmetric matrix
p?syr2k	s, d, c, z	Rank-2k update of a distributed symmetric matrix
p?tran	s, d	Transposition of a real distributed matrix
p?tranc	c, z	Transposition of a complex distributed matrix (conjugated)
p?tranu	c, z	Transposition of a complex distributed matrix
p?trmm	s, d, c, z	Distributed matrix-matrix product, one matrix is triangular
p?trsm	s, d, c, z	Solution of a distributed matrix equation, one matrix is triangular