2.3 Array Descriptors and Block-Cyclic Distribution of the Matrices

When performing calculations, a routine needs to know which global data can be found in which processor's local array. This distribution is passed as an array argument, called the array descriptor vector, and is required by each ScaLAPACK routine. Every distributed matrix A has this integer array associated with it that stores information describing exactly how the matrix is distributed across the process grid. This information uniquely determines the mapping of the elements of matrix A onto the local memory of each logical processor. Except for the array elements that define the BLACS context and the leading dimension of the local array A that is storing the local blocks of matrix A, all elements of the descriptor array are global. So an array descriptor is associated with each global array and stores the information required to establish the mapping between each global array entry and its corresponding process and memory location. Also, array descriptor vector contain the mechanism by which each ScaLAPACK routine determines the distribution of global array elements into local arrays owned by each processor. An array descriptor includes:

- The descriptor type.
- The BLACS context.
- The number of rows in the distributed matrix.
- The number of columns in the distributed matrix.
- The row block size.
- The column block size.
- The process row over which the first row of the matrix is distributed.
- The process column over which the first column of the matrix is distributed.
- The leading dimension of the local array storing the local blocks.

Attention! The leading dimension (LD) in case of column major order is equal to m - the number of rows of the matrix (usually for Fortran). The leading dimension (LD) in case of row major order is equal to n - the number of columns of the matrix (usually for C, C++).

Array descriptors are provided for the following type of matrices:

- dense matrices,
- band and tridiagonal matrices,
- out-of-core matrices,

and are differentiated by the *DTYPE_ entry* in the descriptor. The following values of the DESC_(DTYPE_) are valid.

DESC_(DTYPE_)	DESIGNATION
1	Dense matrices
501	Narrow band and tridiagonal coefficient matrices
502	Narrow band and tridiagonal right-hand-side matrices
601	Out-of-core matrices

The choice of an appropriate data distribution heavily depends on the characteristics or flow of the computation in the algorithm. For dense matrix computations, ScaLAPACK assumes the data to be distributed according to the two-dimensional block-cyclic data layout scheme . The block-cyclic data layout has been selected for the dense algorithms implemented in ScaLAPACK principally because of its scalability, load balance, and efficient use of Level 3 BLAS single processor computation routines (data locality).

The block-partitioned computation proceeds in consecutive order just like a conventional serial algorithm. This essential property of the block cyclic data layout explains why the ScaLAPACK design has been able to reuse the numerical and software expertise of the sequential LAPACK library. Procedure steps of the data distribution method are:

- Divide up the global array into blocks with M_A rows and N_A columns.
- From now on[in viitor], think of the global array as composed only of these blocks.
- Distribute first row of array blocks across the first row of the processor grid in order. If you run out processor grid columns cycle back to first column.
- Repeat for the second row of array blocks, with the second row of the processor grid.
- Continue for remaining rows of array blocks.
- If you run out of processor grid rows, cycle back to the first processor row and repeat.

Example of a block cyclic data distribution

According to the two-dimensional block cyclic data distribution scheme, an M_ by N_ dense matrix is first decomposed into MB_ by NB_ blocks starting at its upper left corner. These blocks are then uniformly distributed in each dimension of the Process Grid.

Thus, every process owns a collection of blocks, which are locally and contiguously stored in a two-dimensional column major array. The partitioning of a matrix into blocks and the mapping of these blocks onto a Process Grid is illustrated with a global 9x9 matrix A. The first step in this process is to partition the matrix A into block. Let us use 2x2 blocks and assume that the 2-D Process Grid is 2x3.

```
a11 a12 | a13 a14 | a15 a16 | a17 a18 | a19
a21 a22 | a23 a24 | a25 a26 | a27 a28 | a29          B11  B12  B13  B14  B15
--------.---------.---------.---------.----
a31 a32 | a33 a34 | a35 a36 | a37 a38 | a39
a41 a42 | a43 a44 | a45 a46 | a47 a48 | a49          B21  B22  B23  B24  B25
--------.---------.---------.---------.----
a51 a52 | a53 a54 | a55 a56 | a57 a58 | a59     ==
a61 a62 | a63 a64 | a65 a66 | a67 a68 | a69          B31  B32  B33  B34  B35
--------.---------.---------.---------.----
a71 a72 | a73 a74 | a75 a76 | a77 a78 | a79
a81 a82 | a83 a84 | a85 a86 | a87 a88 | a89          B41  B42  B43  B44  B45
--------.---------.---------.---------.----
a91 a92 | a93 a94 | a95 a96 | a97 a98 | a99          B51  B52  B53  B54  B55
```

matrix A is M_ x N_ (9 x 9).   A is partitioned into 2x2 blocks. In the above diagram the Bij are the 2x2 blocks, e.g.

```
 B11  ==  a11 a12      B12  ==  a13 a14      ...     B15 == a19
          a21 a22               a23 a24                     a29
```

Initially, the 2x3 Process Grid is empty and looks like this:

	0	1	2
0	.	.	.
1	.	.	.

We identify each process in the Process Grid by two coordinates (row,col). Thus, for the 2x3 Process Grid the processes would be:

	0	1	2
0	(0, 0)	(0, 1)	(0, 2)
1	(1, 0)	(1, 1)	(1, 2)

The distribution process starts by taking the Global B_{ij} in first row and distribute them to the first row of the Processor Grid:

	0	1	2
0	B11 B14	B12 B15	B13
1	.	.	.

Take Global B_{ij} in next row and distribute them to the next row of the Process Grid: (if previous distribution was on last row of Process Grid then restart with row 0 of Process Grid).

	0	1	2
0	B11 B14	B12 B15	B13
1	B21 B24	B22 B25	B23

Take Global B_{ij} in next row and distribute them to the first row of the Process Grid: (restart with row 0 of Process Grid).

	0	1	2
0	B11 B14 B31 B34	B12 B15 B32 B35	B13 B33
1	B21 B24	B22 B25	B23

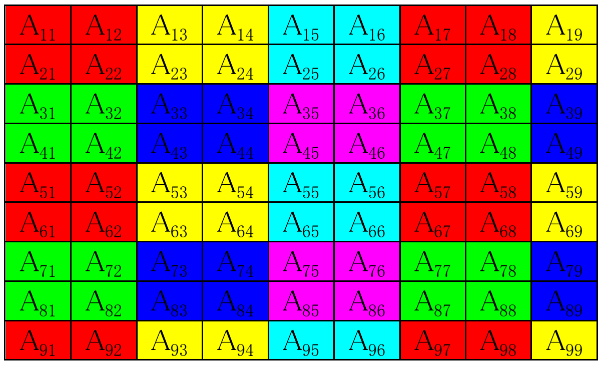
Take Global B_{ij} in next row and distribute them to the next row of the Process Grid:

	0	1	2
0	B11 B14 B31 B34	B12 B15 B32 B35	B13 B33
1	B21 B24 B41 B44	B22 B25 B42 B45	B23 B43

Take Global B_{ij} in next row and distribute them to the first row of the Process Grid: (restart with row 0 of Process Grid).

	0	1	2
0	B11 B14 B31 B34 B51 B54	B12 B15 B32 B35 B52 B55	B13 B33 B53
1	B21 B24 B41 B44	B22 B25 B42 B45	B23 B43

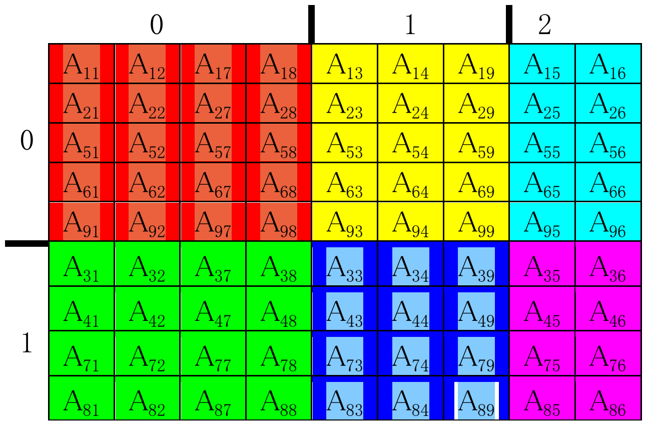
The diagram on the next Figures illustrates a 2-D block-cyclic distribution of a 9x9 global array with 2x2 blocks over a 2x3 processor grid (The colors represent the 6 different processors).



A_{11}	A_{12}	A_{13}	A_{14}	A_{15}	A_{16}	A_{17}	A_{18}	A_{19}
A_{21}	A_{22}	A_{23}	A_{24}	A_{25}	A_{26}	A_{27}	A_{28}	A_{29}
A_{31}	A_{32}	A_{33}	A_{34}	A_{35}	A_{36}	A_{37}	A_{38}	A_{39}
A_{41}	A_{42}	A_{43}	A_{44}	A_{45}	A_{46}	A_{47}	A_{48}	A_{49}
A_{51}	A_{52}	A_{53}	A_{54}	A_{55}	A_{56}	A_{57}	A_{58}	A_{59}
A_{61}	A_{62}	A_{63}	A_{64}	A_{65}	A_{66}	A_{67}	A_{68}	A_{69}
A_{71}	A_{72}	A_{73}	A_{74}	A_{75}	A_{76}	A_{77}	A_{78}	A_{79}
A_{81}	A_{82}	A_{83}	A_{84}	A_{85}	A_{86}	A_{87}	A_{88}	A_{89}
A_{91}	A_{92}	A_{93}	A_{94}	A_{95}	A_{96}	A_{97}	A_{98}	A_{99}

Table "Global View"

Using the procedure steps of the data distribution method are described above we obtain the following local distributed matrix.



	0				1			2	
0	A_{11}	A_{12}	A_{17}	A_{18}	A_{13}	A_{14}	A_{19}	A_{15}	A_{16}
	A_{21}	A_{22}	A_{27}	A_{28}	A_{23}	A_{24}	A_{29}	A_{25}	A_{26}
	A_{51}	A_{52}	A_{57}	A_{58}	A_{53}	A_{54}	A_{59}	A_{55}	A_{56}
	A_{61}	A_{62}	A_{67}	A_{68}	A_{63}	A_{64}	A_{69}	A_{65}	A_{66}
	A_{91}	A_{92}	A_{97}	A_{98}	A_{93}	A_{94}	A_{99}	A_{95}	A_{96}
1	A_{31}	A_{32}	A_{37}	A_{38}	A_{33}	A_{34}	A_{39}	A_{35}	A_{36}
	A_{41}	A_{42}	A_{47}	A_{48}	A_{43}	A_{44}	A_{49}	A_{45}	A_{46}
	A_{71}	A_{72}	A_{77}	A_{78}	A_{73}	A_{74}	A_{79}	A_{75}	A_{76}
	A_{81}	A_{82}	A_{87}	A_{88}	A_{83}	A_{84}	A_{89}	A_{85}	A_{86}

Table "Local (distributed) View"

The array descriptor DESC_A (ScaLPACK notation "_A" read as "of the distributed global array A) for dense matrices is an integer array of length 9. It is used for the ScaLAPACK routines solving dense linear systems and eigenvalue problems. The content of the array descriptor for dense matrices is presented in the following table:

DESC_A()	Symbolic Name	Scope	Destination
1	DTYPE_A	global	Descriptor type DTYPE_A=1 for dense matrices.
2	CTXT_A	global	BLACS context handle, indicating the BLACS process grid over which the global matrix A is distributed. The context itself is global, but the handle (the integer value) may vary.
3	M_A	global	Number of rows in the global array A.
4	N_A	global	Number of columns in the global array A.
5	MB_A	global	Number of rows in the global array A used to distribute the rows of this array.
6	NB_A	global	Number of columns in the global array A used to distribute the

			columns of these array
7	RSRC_A	global	Processor grid row which has the first block of A (typically 0)
8	CSRC_A	global	Processor grid column which has the first block of A (typically 0)
9	LLD_A	local	Total number of rows(Fortran) or columns(C,C++) of the local array that stores the blocks of A (Local Leading Dimension). LLD is (obviously) set at the declaration of the local array. This value of LLD can be different for different processors.

Table "Array descriptor for dense matrices"

For band and tridiagonal matrices, the content of the array descriptor is presented in the following table.

DESC_A()	Symbolic Name	Scope	Destination
1	DTYPE_A	global	Descriptor type DTYPE_A=501 for 1xPc process grid for band and tridiagonal matrices bloc-column distributed.
2	CTXT_A	global	BLACS context handle, indicating the BLACS process grid over which the global matrix A is distributed. The context itself is global, but the handle (the integer value) may vary.
3	N_A	global	Number of columns in the global array A.
4	NB_A	global	Number of columns in the global array A used to distribute the columns of these array
5	CSRC_A	global	Processor grid column which has the first block of A (typically 0)
6	LLD_A	local	Number of rows of the local array that stores the blocks of A (local leading dimension). For the tridiagonal subroutines, this entry is ignored
7			Unused, reserved

Table "Array descriptor for band and tridiagonal matrices"

The ScaLAPACK software library provides routines for solving out-of-core linear systems, in which case the matrices are stored on disk. For out-of-core matrices the content of the array descriptors is presented in the following table

DESC_A()	Symbolic Name	Scope	Destination
1	DTYPE_A	global	Descriptor type DTYPE_A=601 for an out-of-core matrix.
2	CTXT_A	global	BLACS context handle, indicating the BLACS process grid over which the global matrix A is distributed. The context itself is global, but the handle (the integer value) may vary.
3	M_A	global	Number of rows in the global array A.
4	N_A	global	Number of columns in the global array A.
5	MB_A	global	Number of rows in the global array A used to distribute the rows of this array.
6	NB_A	global	Number of columns in the global array A used to distribute the columns of these array
7	RSRC_A	global	Processor grid row which has the first block of A (typically 0)
8	CSRC_A	global	Processor grid column which has the first block of A (typically 0)
9	LLD_A	global	number of rows of the local array that stores the blocks of A (local leading dimension)
10	IODEV_A	global	I/O unit device number associated with the out-of-core matrix A
11	SIZE_A	local	Amount of local in-core memory available for the factorization of A

Table "Array descriptor for out-of-core dense matrices"

Fortunately, you never have to explicitly assign values to each element of DESCA yourself. ScaLAPACK provides the tool routine DESCINIT (DESCriptor INITitalization) that uses its arguments to create the array descriptor vector DESC_A. Of course, DESCINIT must be called by each processor with the appropriate local values. So, just need to call the DESCINIT routine which will create the descriptor vector from is arguments. The syntax for DESCINIT is:

```
DESCINIT(DESC,M,N,MB,NB,RSRC,CSRC,ICONTXT,LLD,INFO)
void descinit_(int*,int*,int*,int*,int*,int*,int*,int*,int*,int*);
```

`DESC (output) INTEGER`

DESC is the "filled-in" descriptor vector returned by the routine.
Arguments 2 through 8 are values for elements 2 through 9 of the descriptor vector (slightly different ordering).

`INFO (output) INTEGER`

These argument is the status value returned to indicate if DESCINIT worked correctly. If INFO=0, the routine call was successful. IF INFO=-i, the i-th argument had an illegal value.

By using the descriptor of the matrices, a call to a PBLAS routine is very similar to a call to the corresponding BLAS routine. For example:

```
    CALL DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,
               A(IA,JA),LDA,
               B(IB,JB),LDB,
               BETA,C(IC,JC),LDC)

    CALL PDGEMM(TRANSA,TRANSB,M,N,K,ALPHA,
               A,IA,JA,DESC_A,
               B,IB,JB,DESC_B,
               BETA,C,IC,JC,DESC_C)
```

DGEMM computes C=BETA×C+ALPHA×op(A)×op(B), where op(A) is either A or its transpose depending on TRANSA, op(B) is similar, op(A) is M-by-K, and op(B) is K-by-N. PDGEMM is the same, with the exception of the way submatrixes are specified. To pass the submatrix starting at A(IA,JA) to DGEMM, for example, the actual argument corresponding to the formal argument A is simply A(IA,JA). PDGEMM, on the other hand, needs to understand the global storage scheme of A to extract the correct submatrix, so IA and JA must be passed in separately.

The ScaLAPACK ensure same Data Distribution Tool Routines by means of which every processor can answer to the following questions:

- How many rows should be in my local array?
- How many columns should be in my local array?
- What global elements should I put in my local array?

ScaLAPACK function NUMROC (Number of Rows Or Columns) will answer the first two questions and the function INDXG2P (Index: global to processor) will answer the third questions. So the NUMROC utility function does the following:

- Returns the number of rows or columns of a local array containing blocks of a distributed global array;
- Will show the arguments for computing the local number of rows. Just switch row to column everywhere to get the number of local columns;
- Returned values are dependent on the calling process.

The syntax for these function are:

```
INTEGER FUNCTION NUMROC(M_,MB,MYROW,RSRC,NPROW)
```

```
int numroc_(int*, int*, int*, int*, int*);
```

The arguments mean the following: M_=number of rows (or columns) in the global array; MB=number of rows (or columns) in each block; MYROW=row (or column) coordinate of the calling processor; RSRC=row (or column) coordinate of the processor containing the first block; NPROW=number of rows (or columns) in the processor grid.

Let K be the number of rows or columns of a distributed matrix, and assume that its process grid has dimension pxq. Then LOCr(K) denotes the number of elements of K that a process would receive if K were distributed over the p processes of its process column. So, the values of LOCr() and LOCc() may be determined via a call to the ScaLAPACK tool function, NUMROC:

LOCr(M)=NUMROC(M,MB_A,MYROW,RSRC_A,NPROW),
LOCc(N)=NUMROC(N,NB_A,MYCOL,CSRC_A,NPCOL).

The INDXG2P utility routine can do the following:

- Given the global indices (i,j) of a certain element of a global array, returns the processor grid coordinates (p,q) that element was distributed to;
- Will show the arguments to INDXG2P in which i is provided to the routine and p is returned. To get q, substitute j for i, and column for row.

The syntax for these function are:

```
INTEGER FUNCTION INDXG2P(INDXGLOB,NB,IPROC,ISRCPROC,NPROCS)
int indxg2p_(int*, int*, int*, int*, int*);
```

Purpose: INDXG2P computes the process coordinate which possesses the entry of a distributed matrix specified by a global index INDXGLOB.

`INDXGLOB (global input) INTEGER`

The global index of the element.

`NB (global input) INTEGER`

Block size, size of the blocks the distributed matrix is split into.

`IPROC (local dummy) INTEGER`

Dummy argument in this case in order to unify the calling sequence of the tool-routines.

`ISRCPROC (global input) INTEGER`

The coordinate of the process that possesses the first row/column of the distributed matrix.

`NPROCS (global input) INTEGER`

The total number processes over which the matrix is distributed.

```
INTEGER FUNCTION INDXL2G(INDXLOC,NB,IPROC,ISRCPROC,NPROCS)
int indxl2g_(int*, int*, int*, int*, int*);
```

Purpose INDXL2G computes the global row or column index of a distributed matrix entry pointed to by the local index INDXLOC of the process indicated by IPROC.

`INDXLOC (global input) INTEGER`

The local index of the distributed matrix entry.

`NB (global input) INTEGER`

Block size, size of the blocks the distributed matrix is split into.

`IPROC (local input) INTEGER`

The coordinate of the process whose local array row or column is to be determined.

`ISRCPROC (global input) INTEGER`

The coordinate of the process that possesses the first row/column of the distributed matrix.

NPROCS (global input) INTEGER

The total number processes over which the distributed matrix is distributed.

Example 2.3.1. *Following program in C++ illustrates the use of these two utility routines for the 9x9 array distributed onto the 2x3 processor grid shown in the Figure "Local (distributed) View" and print the process coordinate that contain the diagonal elements of the distributed matrices.*

```
/* ==========
   This program illustrates the modalities for determination of those elements of the
   matrix, that will be distributed to processors according to the algorithm "two-
   dimensional block-cyclic data layout scheme" The global matrix is not initialized.
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <cmath>
extern "C" {
// Cblacs declarations
     void Cblacs_pinfo(int*,int*);
     void Cblacs_get(int,int,int*);
     void Cblacs_gridinit(int*,const char*,int,int);
      void Cblacs_gridinfo(int,int*,int*,int*,int*);
     void Cblacs_gridexit(int);
     void Cblacs_exit(int);
      int numroc_(int*, int*, int*, int*, int*);
     void Cblacs_barrier(int,const char*);
     int indxg2p_(int*, int*, int*, int*, int*);
             }
int main(int argc, char **argv)
{
int ICTXT,MYID,NPROCS,NPROW,NPCOL;
int P,Q,MYROW,MYCOL,L,K,N,M,Nb,Mb;
int iZERO = 0;
Cblacs_pinfo(&MYID,&NPROCS);
Cblacs_get(-1, 0, &ICTXT);
N=9;M=9;
Nb=2;Mb=2;        //dimensions of the bloc
NPROW=2; NPCOL=3; // dimensions of the process grid
Cblacs_gridinit(&ICTXT, "Row-major", NPROW,NPCOL);
Cblacs_gridinfo(ICTXT,&NPROW,&NPCOL,&MYROW,&MYCOL);
//find out the size of the local array for proc(0,0)
if ((MYROW==0)&(MYCOL==0))
{
   printf("============ REZULT OF THE PROGRAM %s \n",argv[0]);
   printf("The (%d,%d) process owns the %d rows  and %d  cols
  \n",MYROW,MYCOL,numroc_(&N,&Nb,&MYROW,&iZERO,&NPROW),
          numroc_(&M,&Mb,&MYCOL,&iZERO,&NPCOL));
   printf("== THE GLOBAL MATRIX ARE DISTRIBED: \n");
   for (K = 1; K <= N; ++K)
   {
   for (L = 1; L <= M; ++L)
      {
      P=indxg2p_(&K,&Nb,0,&iZERO,&NPROW);
      Q=indxg2p_(&L,&Mb,0,&iZERO,&NPCOL);
      printf("ELEMENT (%d,%d) IS ON PROCES (%d,%d) \n",K,L,P,Q);
      }
   }
}
Cblacs_gridexit(ICTXT);
Cblacs_exit(0);
}
```

The result of the execution of the program

```
[MI_gr_TPS1@hpc]$./mpiCC_ScL -o Example2.3.1.exe Example2.3.1.cpp
[MI_gr_TPS1@hpc]$/opt/openmpi/bin/mpirun -n 6 -host compute-0-10,compute-0-12 Example2.3.1.exe
============ REZULT OF THE PROGRAM Example2.3.1.exe
The (0,0) process owns the 5 rows  and 4  cols
== THE GLOBAL MATRIX ARE DISTRIBED:
ELEMENT (1,1) IS ON PROCES (0,0)
ELEMENT (1,2) IS ON PROCES (0,0)
ELEMENT (1,3) IS ON PROCES (0,1)
ELEMENT (1,4) IS ON PROCES (0,1)
ELEMENT (1,5) IS ON PROCES (0,2)
ELEMENT (1,6) IS ON PROCES (0,2)
ELEMENT (1,7) IS ON PROCES (0,0)
ELEMENT (1,8) IS ON PROCES (0,0)
ELEMENT (1,9) IS ON PROCES (0,1)
ELEMENT (2,1) IS ON PROCES (0,0)
ELEMENT (2,2) IS ON PROCES (0,0)
ELEMENT (2,3) IS ON PROCES (0,1)
ELEMENT (2,4) IS ON PROCES (0,1)
ELEMENT (2,5) IS ON PROCES (0,2)
ELEMENT (2,6) IS ON PROCES (0,2)
ELEMENT (2,7) IS ON PROCES (0,0)
ELEMENT (2,8) IS ON PROCES (0,0)
ELEMENT (2,9) IS ON PROCES (0,1)
ELEMENT (3,1) IS ON PROCES (1,0)
ELEMENT (3,2) IS ON PROCES (1,0)
ELEMENT (3,3) IS ON PROCES (1,1)
ELEMENT (3,4) IS ON PROCES (1,1)
ELEMENT (3,5) IS ON PROCES (1,2)
ELEMENT (3,6) IS ON PROCES (1,2)
ELEMENT (3,7) IS ON PROCES (1,0)
ELEMENT (3,8) IS ON PROCES (1,0)
ELEMENT (3,9) IS ON PROCES (1,1)
ELEMENT (4,1) IS ON PROCES (1,0)
ELEMENT (4,2) IS ON PROCES (1,0)
ELEMENT (4,3) IS ON PROCES (1,1)
ELEMENT (4,4) IS ON PROCES (1,1)
ELEMENT (4,5) IS ON PROCES (1,2)
ELEMENT (4,6) IS ON PROCES (1,2)
ELEMENT (4,7) IS ON PROCES (1,0)
ELEMENT (4,8) IS ON PROCES (1,0)
ELEMENT (4,9) IS ON PROCES (1,1)
ELEMENT (5,1) IS ON PROCES (0,0)
ELEMENT (5,2) IS ON PROCES (0,0)
ELEMENT (5,3) IS ON PROCES (0,1)
ELEMENT (5,4) IS ON PROCES (0,1)
ELEMENT (5,5) IS ON PROCES (0,2)
ELEMENT (5,6) IS ON PROCES (0,2)
ELEMENT (5,7) IS ON PROCES (0,0)
ELEMENT (5,8) IS ON PROCES (0,0)
ELEMENT (5,9) IS ON PROCES (0,1)
ELEMENT (6,1) IS ON PROCES (0,0)
ELEMENT (6,2) IS ON PROCES (0,0)
ELEMENT (6,3) IS ON PROCES (0,1)
ELEMENT (6,4) IS ON PROCES (0,1)
ELEMENT (6,5) IS ON PROCES (0,2)
ELEMENT (6,6) IS ON PROCES (0,2)
ELEMENT (6,7) IS ON PROCES (0,0)
ELEMENT (6,8) IS ON PROCES (0,0)
ELEMENT (6,9) IS ON PROCES (0,1)
ELEMENT (7,1) IS ON PROCES (1,0)
ELEMENT (7,2) IS ON PROCES (1,0)
ELEMENT (7,3) IS ON PROCES (1,1)
ELEMENT (7,4) IS ON PROCES (1,1)
ELEMENT (7,5) IS ON PROCES (1,2)
ELEMENT (7,6) IS ON PROCES (1,2)
ELEMENT (7,7) IS ON PROCES (1,0)
ELEMENT (7,8) IS ON PROCES (1,0)
ELEMENT (7,9) IS ON PROCES (1,1)
ELEMENT (8,1) IS ON PROCES (1,0)
ELEMENT (8,2) IS ON PROCES (1,0)
ELEMENT (8,3) IS ON PROCES (1,1)
ELEMENT (8,4) IS ON PROCES (1,1)
ELEMENT (8,5) IS ON PROCES (1,2)
ELEMENT (8,6) IS ON PROCES (1,2)
ELEMENT (8,7) IS ON PROCES (1,0)
ELEMENT (8,8) IS ON PROCES (1,0)
ELEMENT (8,9) IS ON PROCES (1,1)
ELEMENT (9,1) IS ON PROCES (0,0)
ELEMENT (9,2) IS ON PROCES (0,0)
ELEMENT (9,3) IS ON PROCES (0,1)
ELEMENT (9,4) IS ON PROCES (0,1)
ELEMENT (9,5) IS ON PROCES (0,2)
ELEMENT (9,6) IS ON PROCES (0,2)
ELEMENT (9,7) IS ON PROCES (0,0)
ELEMENT (9,8) IS ON PROCES (0,0)
ELEMENT (9,9) IS ON PROCES (0,1)
```

Example 2.3.2. *This example illustrates the modalities for determination of the "global" matrix elements that will correspond to "local" matrix elements.*

```
     #include <string.h>
     #include <stdlib.h>
     #include <cmath>
     #include "mpi.h"
     extern "C" {
     // Cblacs declarations
          void Cblacs_pinfo(int*,int*);
          void Cblacs_get(int,int,int*);
          void Cblacs_gridinit(int*,const char*,int,int);
                void Cblacs_gridinfo(int,int*,int*,int*,int*);
          void Cblacs_gridexit(int);
          void Cblacs_exit(int);
                int numroc_(int*, int*, int*, int*, int*);
          void Cblacs_barrier(int,const char*);
          int indxl2g_(int*, int*, int*, int*, int*);
                }
     int main(int argc, char **argv)
     {
     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);
     int ictxt, myrow, mycol;
     int info, itemp;
     int ZERO = 0, ONE = 1;
     int iLRow, jLCol, iGRow, jGCol;
     int llda, lldb, lldc;
     int nprow = 2, npcol = 3;
     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 = 9, n = 9;
     int mb = 2, nb = 2;
     int rsrc = 0, csrc = 0;
     // Determine local dimensions : np-by-nq.
     int np = numroc_(&m, &mb, &myrow, &ZERO, &nprow);
     int nq = numroc_(&n, &nb, &mycol, &ZERO, &npcol);
     if ((myrow==0)&(mycol==0))
     {
     printf("The (%d,%d) process owns the %d rows  and %d  cols \n", myrow, mycol, np, nq);
     for (iLRow = 1; iLRow <= np; iLRow++)
     for (jLCol = 1; jLCol <= nq; jLCol++)
         {
     int fortidl = iLRow + 1;
     int fortjdl = jLCol + 1;
     iGRow = indxl2g_(&fortidl, &mb, &myrow, &ZERO, &nprow)-1;
     jGCol = indxl2g_(&fortjdl, &nb, &mycol, &ZERO, &npcol)-1;
     printf("For (%d,%d) proc local index (%d,%d) corespond (%d,%d) global index\n",
             myrow,mycol,iLRow, jLCol, iGRow, jGCol);
         }
     }
     Cblacs_gridexit(0);
     MPI_Finalize();
     return 0;
     }
```

The result of the execution of the program

```
[MI_gr_TPS1@hpc]$./mpiCC_ScL -o Example2.3.2.exe Example2.3.2.cpp
[MI_gr_TPS1@hpc]$ /opt/openmpi/bin/mpirun -n 6 -host compute-0-0,compute-0-1 Example2.3.2.exe

     The (0,0) process owns the 5 rows and 4  cols
For (0,0) proc local index (0,0) corespond (0,0) global index
For (0,0) proc local index (0,1) corespond (0,1) global index
For (0,0) proc local index (0,2) corespond (0,6) global index
For (0,0) proc local index (0,3) corespond (0,7) global index
For (0,0) proc local index (1,0) corespond (1,0) global index
For (0,0) proc local index (1,1) corespond (1,1) global index
For (0,0) proc local index (1,2) corespond (1,6) global index
For (0,0) proc local index (1,3) corespond (1,7) global index
For (0,0) proc local index (2,0) corespond (4,0) global index
For (0,0) proc local index (2,1) corespond (4,1) global index
For (0,0) proc local index (2,2) corespond (4,6) global index
For (0,0) proc local index (2,3) corespond (4,7) global index
For (0,0) proc local index (3,0) corespond (5,0) global index
For (0,0) proc local index (3,1) corespond (5,1) global index
For (0,0) proc local index (3,2) corespond (5,6) global index
For (0,0) proc local index (3,3) corespond (5,7) global index
For (0,0) proc local index (4,0) corespond (8,0) global index
For (0,0) proc local index (4,1) corespond (8,1) global index
For (0,0) proc local index (4,2) corespond (8,6) global index
For (0,0) proc local index (4,3) corespond (8,7) global index
```

It can be checked that the results correspond to the Table “Global View”.

Example 2.3.3. *This example illustrates the modalities for determination of local submatrices (for processes) when the division of matrix is made according to 2-D cyclic algorithm.*

```
     #include <stdio.h>
     #include <string.h>
     #include <stdlib.h>
     #include <iostream>
     #include <iomanip>
     #include <fstream>
     #include <sstream>
     #include <sys/time.h>
     #include "mpi.h"
     using namespace std;
     #define AA(i,j) AA[(i)*n+(j)]
     #define A(i,j) A[(i)*n+(j)]
     extern "C" {
     // Cblacs declarations
          void Cblacs_pinfo(int*,int*);
          void Cblacs_get(int,int,int*);
          void Cblacs_gridinit(int*,const char*,int,int);
```

```
                void Cblacs_gridinfo(int,int*,int*,int*,int*);
          void Cblacs_gridexit(int);
          void Cblacs_exit(int);
                int numroc_(int*, int*, int*, int*, int*);
          void Cblacs_barrier(int,const char*);
          int indxl2g_(int*, int*, int*, int*, int*);
                }
    int main(int argc, char **argv)
    {
    int myrank_mpi, nprocs_mpi;
    double *AA,*A;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank_mpi);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs_mpi);
    int ictxt, myrow, mycol;
    int info, itemp,i,j;
    int ZERO = 0, ONE = 1;
    int iLRow, jLCol, iGRow, jGCol;
    int llda, lldb, lldc;
    int nprow = 2, npcol = 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 = 6, n = 4;
    int mb = 2, nb = 2;
    int rsrc = 0, csrc = 0;
    AA = (double*)malloc(m*n*sizeof(double));
    AA(0,0)=1.44;  AA(0,1)=-7.84; AA(0,2)=-4.39; AA(0,3)=4.53;
    AA(1,0)=-9.96; AA(1,1)=-0.28; AA(1,2)=-3.24; AA(1,3)=3.83;
    AA(2,0)=-7.55; AA(2,1)=3.24;  AA(2,2)=6.27;  AA(2,3)=-6.64;
    AA(3,0)=8.34;  AA(3,1)=8.09;  AA(3,2)=5.28;  AA(3,3)=2.06;
    AA(4,0)=7.08; AA(4,1)=2.52;  AA(4,2)=0.74; AA(4,3)=-2.47;
    AA(5,0)=-5.45;AA(5,1)=-5.70; AA(5,2)=-1.19;AA(5,3)=4.70;
    // Determine local dimensions : np-by-nq.
    int np = numroc_(&m, &mb, &myrow, &ZERO, &nprow);
    int nq = numroc_(&n, &nb, &mycol, &ZERO, &npcol);
    A = new double[np*nq];
    // Construct the submatrices
    for (iLRow =0; iLRow < np; iLRow++)
    for (jLCol =0; jLCol <nq; jLCol++)
    {
    int fortidl = iLRow + 1;
    int fortjdl = jLCol + 1;
    iGRow = indxl2g_(&fortidl, &mb, &myrow, &ZERO, &nprow)-1;
    jGCol = indxl2g_(&fortjdl, &nb, &mycol, &ZERO, &npcol)-1;
    A[iLRow*nq+jLCol]=AA(iGRow,jGCol);
        }
    //Print the submatrices
    Cblacs_barrier(ictxt, "All");
    printf("For (%d,%d) proc local local matrix is\n",myrow,mycol);
    for (i = 0; i < np; ++i)
                   {
      for (j = 0; j < nq; ++j)
      cout << setw(5) << *(A+nq*i+j) << "   ";
      cout << endl;
                   }
    Cblacs_barrier(ictxt, "All");
    Cblacs_gridexit(0);
    MPI_Finalize();
    return 0;
    }
```

The result of the execution of the program

```
[MI_gr_TPS1@hpc]$./mpiCC_ScL -o Example2.3.3.exe Example2.3.3.cpp
[MI_gr_TPS1@hpc]$ /opt/openmpi/bin/mpirun -n 4 -host compute-0-4, Example2.3.3.exe
 For (0,0) proc local local matrix is
 1.44   -7.84
-9.96   -0.28
 7.08    2.52
-5.45    -5.7
For (0,1) proc local local matrix is
-4.39    4.53
-3.24    3.83
 0.74   -2.47
-1.19     4.7
For (1,0) proc local local matrix is
-7.55    3.24
 8.34    8.09
For (1,1) proc local local matrix is
 6.27   -6.64
 5.28    2.06
```

After a global vector or matrix has been block-cyclically distributed over a process grid, the user may choose to perform an operation on a portion of the global matrix. This subset of the global matrix is referred to as a ''submatrix'' and is referenced through the use of six arguments in the calling sequence: the number of rows of the submatrix M, the number of columns of the submatrix N, the local array A containing the global array, the row index IA, the column index JA and the array descriptor of the global array DESCA. This argument convention allows for a global view of the matrix operands and the global addressing of distributed matrices as illustrated in Figure 3. This scheme allows the complete specification of the submatrix A(IA:IA+M-1,JA:JA+N-1) on which to be operated. The description of a global dense subarray consists of (M,N,A,IA,JA,DESCA) that consist of:

- the number of rows and columns M and N of the global subarray,
- a pointer to the local array containing the entire global array (A, for example),
- the row and column indices, (IA, JA), in the global array,
- the array descriptor, DESCA, for the global array.

The names of the row and column indices for the global array have the form I<array_name> and J<array_name> ,respectively.

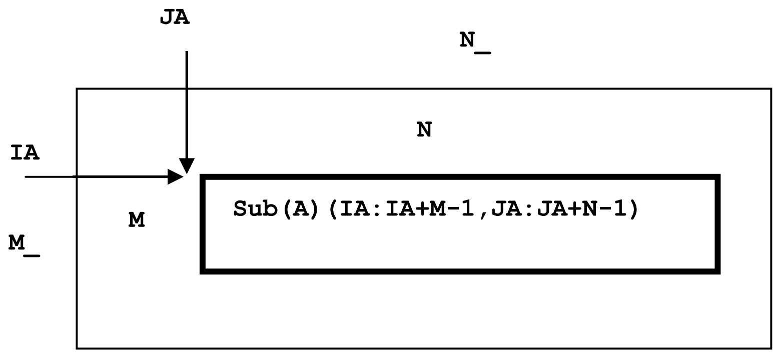


Figure 3

So for using the subroutines of LAPACK/BLAS and ScaLAPACK/PBLAS will used the following subarray syntax.

a) A long-standing design feature of single-processor LAPACK/BLAS routines has been the ability of the user to only use a section of an array - a subarray- in the call to a calculation subroutine.
    - The entire array must still be declared and initialized before the library call.

- The syntax for specifying a subarray has always followed the same pattern.

b) CALL LAPACK_routine (…, M, N, A(I,J), …).
   - This trio of arguments tells the routine do not use the entire array A, but use a subarrray starting at element A(I,J) and extending for M rows and N columns.

c) The developers of the parallel ScaLAPACK/PBLAS libraries decided to also allow users the ability to use subarrays as the operands for parallel calculations. But the syntax is different.
   - All the preparatory steps must still being carried out before the call to the ScaLAPACK routines.
   - Syntax for ScaLAPACK routines now involves the following arguments.

d) CALL ScaLPACK_routine (…, M, N, LA, IA, JA, DESCA, …).
   - These arguments indicate to use a subarray of the global array A starting at element A(IA,JA) and extending for M rows and N columns.
   - Notice that the actual array argument - LA - is still the local array holding blocks of the global array A and that the array descriptor vector is (as always) needed.

The next ScaLAPACK tools routine, named INFOG2L, computes the starting local indexes LRINDX, LCINDX corresponding to the distributed submatrix starting globally at the entry pointed by GRINDX, GCINDX. This routine returns the coordinates in the grid of the process owning the matrix entry of global indexes GRINDX, GCINDX, namely RSRC and CSRC.

The syntaxes of the routine INFOG2L is presented below.

```
INFOG2L(GRINDX,GCINDX,DESC,NPROW,NPCOL,MYROW,MYCOL,LRINDX,
        LCINDX,RSRC,CSRC)
int infog2l_(int*,int*,int*,int*,int*,int*,int*,int*,int*,int*,int*);
```

`GRINDX (global input) INTEGER`
: *The global row starting index of the submatrix.*

`GCINDX (global input) INTEGER`
: *The global column starting index of the submatrix.*

`DESC (input) INTEGER array of dimension DLEN_.`
: *The array descriptor for the underlying distributed matrix.*

`NPROW (global input) INTEGER`
: *The total number of process rows over which the distributed matrix is distributed.*

`NPCOL (global input) INTEGER`
: *The total number of process columns over which the distributed matrix is distributed.*

`MYROW (local input) INTEGER`
: *The row coordinate of the process calling this routine.*

`MYCOL (local input) INTEGER`
: *The column coordinate of the process calling this routine.*

`LRINDX (local output) INTEGER`
: *The local rows starting index of the submatrix.*

`LCINDX (local output) INTEGER`
: *The local columns starting index of the submatrix.*

`RSRC (global output) INTEGER`

The row coordinate of the process that possesses the first row and column of the submatrix.

`CSRC (global output) INTEGER`

The column coordinate of the process that possesses the first row and column of the submatrix.

You must not confusing INFOG2L with the subroutine pair INXG2L/INDXG2P. The purpose of INFOG2L is: consider a distributed matrix A(1:M,1:N), and consider the submatrix A(GRINDX:M,GCINDX:N) where GRINDX,GCINDX are global indices. Each process will own a (possibly empty) different "slice" of this subarray; and on each process this "slice" will start at specific indices (LRINDX,LCINDX), which are exactly the output of INFOG2L. RSRC, and CSRC will return the same values to all processes, identifying the process holding the very first entry A(GRINDX,GCINDX) of the submatrix.

Example 2.3.4. *This example illustrates the arrangements for using the utility infog2l_.*

```
#include <string.h>
#include <stdlib.h>
#include <cmath>
#include "mpi.h"
#include <iomanip>
#include <fstream>
#include <iostream>
#include <sstream>
#include <cstdlib>
extern "C" {
// Cblacs declarations
     void Cblacs_pinfo(int*,int*);
     void Cblacs_get(int,int,int*);
     void Cblacs_gridinit(int*,const char*,int,int);
     void Cblacs_gridinfo(int,int*,int*,int*,int*);
     void Cblacs_gridexit(int);
     void Cblacs_exit(int);
     int numroc_(int*, int*, int*, int*, int*);
     void Cblacs_barrier(int,const char*);
     void descinit_(int*,int*,int*,int*,int*,int*,int*,int*,int*,int*);
     int infog2l_(int*,int*,int*,int*,int*,int*,int*,int*,int*,int*,int*);
                 }
// function returning the max between two numbers
       int max(int num1, int num2)
       {
         // local variable declaration
         int result;
         if (num1 > num2)
                 result = num1;
              else
                 result = num2;
         return result;
       }
     int main(int argc, char **argv)
     {
     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);
     int descA[9];
     int ictxt, myrow, mycol;
     int info, itemp;
     int ZERO = 0, ONE = 1;
     int grindx,gcindx;
     int nprow = 2, npcol = 3;
     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 = 9, n = 9;// m-rows, n-columns
     int mb = 2, nb = 2;  //mb rows of the bloc, nb column of the bloc
     int rsrc = 0, csrc = 0; //Processor grid row and column which has the first block of A
     int lrindx,lcindx;
     // Determine local dimensions : np-by-nq.
     int np = numroc_(&m, &mb, &myrow, &ZERO, &nprow);
     int nq = numroc_(&n, &nb, &mycol, &ZERO, &npcol);
     int lda = max(1,nq);//max(1,np);
     info=0;
     descinit_(descA, &m, &n, &mb, &nb, &ZERO,&ZERO,&ictxt, &lda, &info);
      if (myrank_mpi==0)
     {
     printf("======== \n");
      if (info == 0){
         printf("Description init sucesss!\n");
           }
           if(info < 0){
        printf("Error Info = %d, if INFO = -i, the i-th argument had an illegal
             value\n",info);
           }
     }
     Cblacs_barrier(ictxt, "All");
     if ((myrow==1)&(mycol==2))
     {
     printf("For process (%d,%d) descA[0]=%d, descA[1]=%d, descA[2]=%d, descA[3]=%d,
       descA[4]=%d, descA[5]=%d,descA[6]=%d,descA[7]=%d, descA[8]=%d. \n", myrow, mycol,
       descA[0], descA[1], descA[2], descA[3], descA[4],descA[5],descA[6],descA[7],
       descA[8]);
     grindx=6; gcindx=6;
    infog2l_(&grindx,&gcindx,descA,&nprow,&npcol,&myrow,&mycol,&lrindx,&lcindx,&rsrc,&csrc);
     printf("The glabal element is (%d,%d) and process (%d,%d) will own it. \n",grindx,
             gcindx, rsrc,csrc);
     printf("Process (%d,%d) will own a slice of this subarray that start at  local indice
       (%d,%d).\n",myrow, mycol,lrindx,lcindx);
     }
Cblacs_gridexit(0);
MPI_Finalize();
return 0;
}
```

The result of the execution of the program

```
[MI_gr_TPS1@hpc]$./mpiCC_ScL Example2.3.4.cpp -o Example2.3.4.exe
[MI_gr_TPS1@hpc]$ /opt/openmpi/bin/mpirun -n 6 -host compute-0-   10,compute-0-12
Example2.3.4.exe
========
Description init sucesss!
For process (1,2) descA[0]=1, descA[1]=0, descA[2]=9, descA[3]=9, descA[4]=2,  descA[5]=2,
descA[6]=0,descA[7]=0, descA[8]=4
The glabal element is (6,6) and process (0,2) will own it.
Process (1,2) will own a slice of this subarray that start at  local indice(3,2).
```

Attention! The function infog2l_ can be used to initialize (to assign the values) the local submatrices on the base of indices of the global matrix. If in the above program it will be added the following fragment.

```
.
.
.
```

```
/ == the use of the function infog21_to assign the values of local submatrices on the base of
indices of the global matrix
double *A_loc = NULL;
A_loc = new double[np*nq];// space reservation for local matrices
for (int r = 0; r < m; ++r)
{
        for (int c = 0; c <n; ++c)
{
if (r==c)
{
infog2l_(&r,&c,descA,&nprow,&npcol,&myrow,&mycol,&lrindx,&lcindx,&rsrc,&csrc);
if ((myrow==rsrc)&(mycol==csrc))
{
A_loc[lrindx,lcindx]=100;
printf("On node (%d,%d) A_loc[%d,%d](global[%d,%d])=%f
\n",myrow,mycol,lrindx,lcindx,r,c,A_loc[lrindx,lcindx]);
//Cblacs_barrier(ictxt, "All");
}
}
}
}
     .
     .
     .
```

then the local matrices there are assigned the values that correspond to the elements on the main diagonal of the global matrix.

```
[MI_gr_TPS1@hpc]$ /opt/openmpi/bin/mpirun -n 6 -host compute-0-0,compute-0-1 Example2.3.5.exe
========
Description init success!
Process (0,0) will own a slice of this subarray that start at  local indice (1,1).
Process (0,2) will own a slice of this subarray that start at  local indice (1,1).
Process (1,0) will own a slice of this subarray that start at  local indice (1,1).
Process (0,1) will own a slice of this subarray that start at  local indice (1,1).
Process (1,2) will own a slice of this subarray that start at  local indice (1,1).
Process (1,1) will own a slice of this subarray that start at  local indice (1,1).
On node (0,0) A_loc[1,1](global[1,1])=100.000000
On node (0,0) A_loc[2,2](global[2,2])=100.000000
On node (1,1) A_loc[1,1](global[3,3])=100.000000
On node (1,1) A_loc[2,2](global[4,4])=100.000000
On node (0,2) A_loc[3,1](global[5,5])=100.000000
On node (0,2) A_loc[4,2](global[6,6])=100.000000
On node (1,0) A_loc[3,3](global[7,7])=100.000000
On node (1,0) A_loc[4,4](global[8,8])=100.000000
On node (0,1) A_loc[5,3](global[9,9])=100.000000
```