2.2 Data Distributions and Software Conventions

The ScaLAPACK software library provides routines that operate on three types of matrices: in-core dense matrices, in-core narrow band matrices and out-of-core dense matrices. On entry, these routines assume that the data has been distributed on the processors according to a specific data decomposition scheme. Conventional[de obicei] arrays are used to store locally the data when it resides in the processors' memory. The data layout[schema] information as well as the local storage scheme for these different matrix operands is conveyed to the routines via a simple array of integers called an array descriptor. The first entry of this array identifies the type of the descriptor, i.e., the data distribution scheme it describes. This paragrath first presents the fundamental concepts of process grids, communication contexts and array descriptors. Then, for each of the three possible matrix operand mentioned above, the data distribution scheme and the corresponding descriptor array used by ScaLAPACK are discussed in detail. Finally, the software conventions common to all ScaLAPACK routines are presented.

Deci, pentru a realiza transmiterea datelor (aici elementele unei matrici) intre procese sunt necesare:

- **generarea unui grid de procese;**
- **descrierea unui mediu de comunicare, unui context de comunicare;**
- **determinarea (initializarea) unui descriptor pentru fiecare matrice.**

2.2.1 Process Grid, Scoped Operations and Context

The P processes of an abstract parallel computer are often represented as a one-dimensional linear array of processes labelled 0,1,…,P. For reasons described below, it is often more convenient to map this one-dimensional array of processes into a two-dimensional rectangular grid, or process grid by using row-major order (the numbering of the processes increases sequentially across each row) or by using column-major order (the numbering of the processes proceeds down each column of the process grid).

This grid will have Pr process rows and Pc process columns, where Pr×Pc=P. A process can now be referenced by its row and column coordinates, (pr,pc), within the grid, where 0≤pr<Pr, and 0≤pc<Pc. An example of such a mapping is shown in the following figure, where Pr=2 and Pc=4:

0	1	2	3
0	1	2	3
4	5	6	7

An operation which involves more than just a sender and a receiver is called a scoped operation. All processes that participate in a scoped operation are said to be within the operation's scope. On a system using a linear array of processes, the only natural scope is all processes. Using a 2D grid, we have 3 natural scopes, as shown in in the following table

Scope	Meaning
Row	All processes in a process row participate.
Column	All processes in a process column participate.
All	All processes in the process grid participate.

These groupings of processes are of particular interest to the linear algebra programmer, since distributed data decompositions of a linear algebra matrix tend to follow this process mapping. Viewing the rows/columns of the process grid as essentially autonomous subsystems provides the programmer with additional levels of parallelism.

The BLACS routine BLACS_GRIDINIT performs the task of mapping the processes to the process grid. By default, BLACS_GRIDINIT assumes a row-major ordering, although a column-major ordering can also be specified. The companion routine BLACS_GRIDMAP is a more general form of BLACS_GRIDINIT and allows the user to define the mapping of the processes.

In ScaLAPACK, and thus the BLACS, each process grid is enclosed in a context. Similarly, a context is associated with every global matrix in ScaLAPACK. The use of a context provides the ability to have separate "universes" of message passing. This means that a process grid can safely communicate even if other process grids are also communicating. Thus, a context is a powerful mechanism for avoiding unintentional nondeterminism in message passing and provides support for the design of safe, modular software libraries. In MPI, this concept is referred to as a communicator.

A context partitions the communication space. A message sent from one context cannot be received in another context. The use of separate communication contexts by distinct libraries (or distinct library routine invocations) insulates communication internal to a specific library routine from external communication that may be going on within the user's program. In most respects, we can use the terms process grid and context interchangeably. So context allows us to do the following:

- *create arbitrary groups of processes,*
- *create an indeterminate number of overlapping and/or disjoint process grids,*
- *isolate the process grid so that they do not interfere with each other.*

In the BLACS, there are two grid creation routines (BLACS_GRIDINIT and BLACS_GRIDMAP) which create a process grid and its enclosing context. These routines return context handles, which are simple integers, assigned by the BLACS to identify the context. Subsequent BLACS routines will be passed these handles, which allow the BLACS to determine from which context/grid a routine is being called. The user should never actually manipulate these handles; they are opaque data objects which are only meaningful for the BLACS routines.

A defined context consumes resources. It is therefore advisable to release contexts when they are no longer needed. This is done via the routine BLACS_GRIDEXIT. When the entire BLACS system is shut down (via a call to BLACS_EXIT), all outstanding contexts are automatically freed.

We describe the BLACS routines that deal with grid/context creation, and processing before the grid/context has been defined

BLACS_PINFO(MYPNUM,NPROCS)
void Cblacs_pinfo(int*,int*);
MYPNUM (output) INTEGER
An integer between 0 and (NPROCS-1) which uniquely identifies each process.
NPROCS (output) INTEGER
The number of processes available for BLACS use.

These routine is used when some initial system information is required before the BLACS are set up. Primary purpose of calling this routine is to get the number of processors that the program will use (specified on parallel run command line) and the ID number of the processor that called the routine.

Atentie! Aceasta rutina este similara functiilor MPI_Comm_size si MPI_Comm_rank.

BLACS_GET(-1,0,ICONTXT)
void Cblacs_get(int,int,int*);
ICONTXT (output) INTEGER

BLACS_GET is a general-purpose values for a set of BLACS internal parameters. With the arguments shown above, the BLACS_GET call will get the context (argument ICONTXT) for ScaLAPACK use. (Note: the first argument is ignored in this form of the call.)

BLACS_GRIDINIT(ICONTXT,ORDER,NPROW,NPCOL)
void Cblacs_gridinit(int*,const char*,int,int);
ICONTXT (input/output) INTEGER
On input, an integer handle indicating the system context to be used in creating the BLACS context. The user may obtain a default system context via a call to BLACS_GET. On output, the integer handle to the created BLACS context.
ORDER (input) CHARACTER*1
Indicates how to map processes to BLACS grid. Choices are: 'R' - use row-major natural ordering; 'C' - use column-major natural ordering; ELSE - use row-major natural ordering.
NPROW (input) INTEGER
Indicates how many process rows the process grid should contain.
NPCOL (input) INTEGER
Indicates how many process columns the process grid should contain.

All codes that use the ScaLAPACK (BLACS) must call this routine, or its companion routine BLACS_GRIDMAP. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system will map into the native machine's process numbering system. Each BLACS grid is contained in a context (its own message passing universe), so that it does not interfere with distributed operations which occur within other grids/contexts. These grid creation routines may be called repeatedly in order to define additional contexts/grids.

The creation of a grid requires input from all processes which are defined to be in it. It is therefore a globally-blocking (sometimes called synchronous) operation which means that processes belonging to more than one grid will have to agree on which grid formation will be serviced first. Note that these routines map already-existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes will be actual processors (hardware), and they are “created" when the user runs his executable. This routine creates a simple NPROW×NPCOL process grid. This process grid will use the first NPROW×NPCOL processes, and assign them to

the grid in a row or column-major natural ordering. If these process-to-grid mappings are unacceptable, more complex companion routine BLACS_GRIDMAP must be called instead.

BLACS_GRIDMAP(ICONTXT,USERMAP,LDUMAP,NPROW,NPCOL)
void Cblacs_gridmap(int*, int*, int, int, int);
ICONTXT (input/output) INTEGER
On input, an integer handle indicating the system context to be used in creating the BLACS context. The user may obtain a default system context via a call to BLACS_GET. On output, the integer handle to the created BLACS context.
USERMAP (input) INTEGER array, dimension (LDUMAP,NPCOL)
Input array indicating the process-to-grid mapping.
LDUMAP (input) INTEGER
The leading dimension of the 2D array USERMAP.
NPROW (input) INTEGER
Indicates how many process rows the process grid should contain.
NPCOL (input) INTEGER
Indicates how many process columns the process grid should contain.

These routines similar the BLACS_GRIDINIT take the available processes, and assign, or map, them into a BLACS process grid. This routine allows the user to map processes to the process grid in an arbitrary manner. USERMAP(i,j) holds the process number of the process to be placed in {i,j} of the process grid. On most distributed systems, this process number will simply by a machine defined number between 0,…,NPROCS-1. BLACS_GRIDMAP is not for the inexperienced user – BLACS_GRIDINIT is much simpler. BLACS_GRIDINIT simply performs a GRIDMAP where the first NPROW*NPCOL processes are mapped into the current grid in a row- or column-major natural ordering. BLACS_GRIDMAP allows the experienced user to take advantage of the processors' actual network (i.e. he can map nodes that are physically connected to be neighbours in the BLACS grid, etc.). BLACS_GRIDMAP also opens the way for multigridding: the user can separate his nodes into arbitrary grids, join them together at some later date, and then re-split them into new grids. BLACS_GRIDMAP also provides the ability to make arbitrary grids or subgrids (e.g., a “nearest neighbour" grid), which can greatly facilitate operations among groups of processes which do not fall on a row or column of the main process grid.

The following routines destroy grids and free resources.

BLACS_FREEBUFF(ICONTXT,WAIT)
void Cblacs_freebuf(int,int);
ICONTXT (input) INTEGER
Integer handle indicating the BLACS context.
WAIT (input) INTEGER
Whether to wait on non-blocking operations:
IF (WAIT .EQ. 0) THEN
Do not wait on operations, free only unused buffers.
ELSE
If necessary, wait in order to free all buffers.
END IF

The BLACS have at least one internal buffer that is used for packing messages (the number of internal buffers varies, depending on which BLACS you are using). On systems where memory is tight, keeping this buffer(s) around may become expensive. Calling this routine will release the BLACS buffer(s). However, the next call to a communication routine which requires packing will cause the buffer to be reallocated. The parameter WAIT determines whether the BLACS should wait for any non-blocking operations to complete or not. If WAIT=0, the BLACS will free any buffers that can be freed without waiting. If WAIT is not 0, the BLACS will free all internal buffers, possibly causing the call to block while the BLACS wait for internal non-blocking operations complete.

BLACS_GRIDEXIT(ICONTXT)
void Cblacs_gridexit(int);
ICONTXT (input) INTEGER
Integer handle indicating the BLACS context to be freed.

Contexts consume resources, and therefore the user should release them when they are no longer needed. BLACS_GRIDEXIT frees a context. After the freeing of a context, the context no longer exists, and its handle may be re-issued by the BLACS if a new context is defined.

BLACS_EXIT(CONTINUE)
void Cblacs_exit(int);

CONTINUE (input) INTEGER.
If CONTINUE is non-zero, it is assume the user will continue using the machine after the BLACS are done. Otherwise, it is assumed that no message passing will be done after the BLACS_EXIT call.

This routine should be called when a process has finished all use of the BLACS. It frees all BLACS contexts and releases all memory the BLACS have allocated. CONTINUE indicates whether the user will be using the underlying communication platform after the BLACS are finished.

The following routines return information involving the process grid. Also included here is the barrier routine.

BLACS_GRIDINFO(ICONTXT,NPROW,NPCOL,MYPROW,MYPCOL)
void Cblacs_gridinfo(int,int*,int*,int*,int*);

ICONTXT (input) INTEGER
Integer handle indicating the BLACS context to be queried.
NPROW (output) INTEGER
On output, the number of process rows in ICONTXT's process grid.
NPCOL (output) INTEGER
On output, the number of process columns in ICONTXT's process grid.
MYPROW (output) INTEGER
On output, the calling process's row coordinate in the process grid.
MYPCOL (output) INTEGER
On output, the calling process's column coordinate in the process grid.

Returns information about the process grid contained in the context whose handle is ICONTXT. If the context handle is invalid, all quantities are returned as -1.

INTEGER FUNCTION BLACS_PNUM(ICONTXT,PROW,PCOL)
int Cblacs_pnum(int, int, int);

ICONTXT (input) INTEGER
Handle indicating the BLACS context to be queried.
PROW (input) INTEGER
The row coordinate of the process whose system process number is to be determined.
PCOL (input) INTEGER
The column coordinate of the process whose system process number is to be determined.

This function returns the system process number of the process at PROW, PCOL in the process grid.

BLACS_PCOORD(ICONTXT,PNUM,PROW,PCOL)
void Cblacs_pcoord(int, int, int*, int*);

ICONTXT (input) INTEGER
Integer handle indicating the BLACS context.
PNUM (input) INTEGER
The process number whose coordinates is to be determined. This is the process number of the underlying machine.
PROW (output) INTEGER
On output, the row coordinate of process PNUM in the BLACS grid.
PCOL (output) INTEGER
On output, the column coordinate of process PNUM in the BLACS grid.

Given the system process number, returns the row and column coordinates in the BLACS' process grid.

A call to the ScaLAPACK TOOLS routine SL_INIT initializes the process grid. This routine initializes a $P_r \acute{} P_c$ (denoted *NPROW´NPCOL* in the source code) process grid by using a row-major ordering of the processes, and obtains a default system *context.* The user can then query the process grid to identify each process's coordinates (*MYROW*,*MYCOL*)) via a call to BLACS_GRIDINFO.

A typical code fragment to accomplish this task would be the following:

```
CALL SL_INIT(ICTXT,NPROW,NPCOL)
CALL BLACS_GRIDINFO(ICTXT,NPROW,NPCOL,MYROW,MYCOL)
```

where details of the calling sequence for SL_INIT are provided below.

SL_INIT(ICTXT,NPROW,NPCOL)

ICTXT (global output) INTEGER
ICTXT specifies the BLACS context identifying the created process grid.
NPROW (global input) INTEGER
NPROW specifies the number of process rows in the process grid to be created.
NPCOL (global input) INTEGER
NPCOL specifies the number of process columns in the process grid to be created.

Example 1.1

Using these routines we can elaborate the program that realizes the steps 2),3) and 8) by means of ScaLAPACK system.

The program of this type in C++ is represented bellow.

```
#include <stdio.h>
#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_pcoord(int, int, int*, int*);
        void Cblacs_gridexit(int);
        void Cblacs_barrier(int, const char*);
        }
int main(int argc, char **argv)
{
// Initializarea
int ICTXT,MYID,NPROCS,NPROW,NPCOL;
int P,Q,MYROW,MYCOL;
int BLACS_PNUM;
int rank, namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Get_processor_name(processor_name,&namelen);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
// initializing the BLACS library (step 2))
Cblacs_pinfo(&MYID,&NPROCS);
Cblacs_get(-1, 0, &ICTXT);
// calculations to make the most square-like proc grid possible
NPROW=(int)sqrt(NPROCS);
NPCOL=NPROCS/NPROW;
// creating and using the processor grid (step 3))
Cblacs_gridinit(&ICTXT, "Row-major", NPROW,NPCOL);
Cblacs_gridinfo(ICTXT,&NPROW,&NPCOL,&MYROW,&MYCOL);
 if ((MYROW==0)&(MYCOL==0))
printf("============ REZULT OF THE PROGRAM %s \n",argv[0]);
```

```
Cblacs_pcoord(ICTXT, rank, &P,&Q);
Cblacs_barrier(ICTXT, "All");
printf("Hello, I am the process number %d on the compute hostname %s , and have a grid position (%d,%d)\n",rank,
processor_name, P,Q);
Cblacs_barrier(ICTXT, "All");
if (rank ==0) printf("=== \n");
Cblacs_barrier(ICTXT, "All");
printf("A grid position (%d,%d) is the process number %d on the compute hostname %s \n",P,Q,rank,
processor_name);
// release the proc gird and blacs library (step 8))
Cblacs_gridexit(ICTXT);
MPI_Finalize();
return 0;
}
```

Result of compilation and executing

```
    [UAS_M31@hpc ScaLAPACK_for_C]$ ./mpiCC_ScL -o Example1.1.exe Example1.1.cpp
    Proces_grid.cpp: In function 'int main(int, char**)':
    Proces_grid.cpp:38: warning: converting to 'int' from 'double'
    [UAS_M31@hpc ScaLAPACK_for_C]$ /opt/openmpi/bin/mpirun -n 9 -host compute-0-10,compute-0-
12,compute-0-2 Example1.1.exe
    ============ REZULT OF THE PROGRAM Proces_grid.exe
    The most square-like proc grid possible is 3x3
    On the node compute-0-10.local processor 0 is at  grid position (0,0)
    On the node compute-0-2.local processor 8 is at  grid position (2,2)
    On the node compute-0-10.local processor 3 is at  grid position (1,0)
    On the node compute-0-2.local processor 2 is at  grid position (0,2)
    On the node compute-0-12.local processor 4 is at  grid position (1,1)
    On the node compute-0-2.local processor 5 is at  grid position (1,2)
    On the node compute-0-12.local processor 1 is at  grid position (0,1)
    On the node compute-0-12.local processor 7 is at  grid position (2,1)
    On the node compute-0-10.local processor 6 is at  grid position (2,0)
    ===
    At grid position (1,0) is processor 3 on the node compute-0-10.local
    At grid position (2,2) is processor 8 on the node compute-0-2.local
    At grid position (1,1) is processor 4 on the node compute-0-12.local
    At grid position (2,0) is processor 6 on the node compute-0-10.local
    At grid position (0,2) is processor 2 on the node compute-0-2.local
    At grid position (2,1) is processor 7 on the node compute-0-12.local
    At grid position (0,0) is processor 0 on the node compute-0-10.local
    At grid position (1,2) is processor 5 on the node compute-0-2.local
    At grid position (0,1) is processor 1 on the node compute-0-12.local
```