2.4 The Basic BLACS communication routines

The BLACS (Basic Linear Algebra Communication Subprograms) are used as the communication layer for the ScaLAPACK project, which involves implementing the LAPACK library on distributed memory Multiple Instruction Stream Multiple Data Stream (MIMD) machines. In this paragraph we will describe the basic routines of the BLACS library.

In the message passing there is three levels of blocking: non-blocking, locally-blocking, and globally-blocking. These levels are briefly previewed below.

Non-blocking communication: the return from the communication routine implies only that the message request has been posted. It is then the user's responsibility to probe and thus determine when the operation has completed.

Locally-blocking communication: may be applied only to send operations, not receives. The return from the send implies that the buffer is available for re-use. It is further specified that the send will complete regardless of whether the corresponding receive is posted.

Globally-blocking communication: the return from the operation implies that the buffer is available for re-use. The operation may not complete unless the complement of the operation is called (e.g., a send may not complete if the corresponding receive is not posted).

The naming conventions for each of the four BLACS routine classifications (point to point communication, broadcast, combine and support) are presented below. Points to point, broadcast and combine are all typed routines, i.e., there is a separate routine for each data type.

!The names of the point to point and broadcast communication routines follow the template vXXYY2D, where the letter in the v position indicates the data type being sent, XX is replaced to indicate the shape[forma] of the matrix, and the YY positions are used to indicate the type of communication to perform. This is shown in Table 4.

v	**MEANING**
I	Integer data is to be communicated
S	Single precision real data is to be communicated.
D	Double precision real data is to be communicated
C	Single precision complex data is to be communicated
Z	Double precision complex data is to be communicated
XX	**MEANING**
GE	The data to be communicated is stored in a general rectangular matrix
TR	The data to be communicated is stored in a trapezoidal matrix.
YY	**MEANING**
SD	Send. One process sends to another
RV	Receive. One process receives from another
BS	Broadcast/send. A process begins the broadcast of data within a scope.
BR	Broadcast/recv. A process receives and participates in the broadcast of data within a scope

Table 4 “Values and meanings of the communication routines' name positions”

!The general form of the names for combines routines is vGZZZ2D, where v is the same as shown in Table 4. The position ZZZ indicates what type of operation should be performed when sending the data. The operations presently supported are shown on Table 5.

ZZZ	MEANING
AMX	Entries of result matrix will have the value of the greatest absolute value found in that position
AMN	Entries of result matrix will have the value of the smallest absolute value found in that position
SUM	Entries of result matrix will have the summation of that position.

Table 5. "Values and meanings of combine routines' name positions"

The support routines serve many diverse functions, and thus they do not have a great degree of standardization. All official BLACS support routines (i.e., those that are guaranteed by the standard to exist) have the form BLACS_<name>.

2.4.1 BLACS Point to point communication routines

Point to point communication requires two complementary operations. The send operation produces a message, which is then consumed by the receive operation. The BLACS send is defined to be locally-blocking, and the receive is globally-blocking. In addition, the BLACS specify that point to point messages between two given processes will be strictly ordered. Therefore, if process 0 sends three messages (label them A, B, and C) to process 1, process 1 must receive A before it can receive B, and message C can be received only after both A and B. It should be noted, however, that messages from different processes are not ordered. Therefore, if processes 0; : : :; 3 send messages A; : : :;D, respectively, to process 4, process 4 may receive these messages in any order that is convenient.

The syntaxes of the Point to Point Sends routines are the following:

```
vGESD2D(ICONTXT,M,N,A,LDA,RDEST,CDEST)
void Cdgesd2d(int, int, int, double*, int, int, int) ;

vTRSD2D(ICONTXT,UPLO,DIAG,M,N,A,LDA,RDEST,CDEST)
void Cdtrsd2d(int,char*,char*,int,int,double*,int,int,int)
```

ICONTXT (input) INTEGER
The BLACS context handle.

UPLO (input) CHARACTER*1
Indicates whether the matrix is upper (UPLO='U') or lower (UPLO='L') trapezoidal.

DIAG (input) CHARACTER*1
Indicates whether the diagonal of the matrix is unit diagonal (DIAG='U'), and thus need not be communicated, or otherwise (DIAG='N').

M (input) INTEGER
The number of matrix rows to be sent.

N (input) INTEGER
The number of matrix columns to be sent.

A (input) TYPE array of dimension (LDA,N)

A pointer to the beginning of the (sub)array to be sent.

`LDA (input) INTEGER`

The leading dimension of the matrix A, i.e., the distance between two successive elements in a matrix row or column.

`RDEST (input) INTEGER`

Process row coordinate of the destination process.

`CDEST (input) INTEGER`

Process column coordinate of the destination process.

The syntaxes of the Point to Point Receives routines are the following:

```
vGERV2D(ICONTXT,M,N,A,LDA,RSRC,CSRC)
void Cdgerv2d(int,int,int,double*,int,int,int);

vTRRV2D(ICONTXT,UPLO,DIAG,M,N,A,LDA,RSRC,CSRC)
void   Cdtrrv2d(int,char*,char*,int,int,double*,int,int,int)
```

`ICONTXT (input) INTEGER`

The BLACS context handle.

`UPLO (input) CHARACTER*1`

Indicates whether the matrix is upper (`UPLO='U'`) or lower (`UPLO='L'`) trapezoidal.

`DIAG (input) CHARACTER*1`

Indicates whether the diagonal of the matrix is unit diagonal (DIAG='U'), and thus need not be communicated, or otherwise (DIAG='N').

`M (input) INTEGER`

The number of matrix rows to be received.

`N (input) INTEGER`

The number of matrix columns to be received.

`A (output) TYPE array (LDA,N)`

A pointer to the beginning of the (sub)array to be received.

`LDA (input) INTEGER`

The leading dimension of the matrix `A`, i.e., the distance between two successive elements in a matrix row.

`RSRC (input) INTEGER`

Process row coordinate of the source of the message.

`CSRC (input) INTEGER`

Process column coordinate of the source of the message.

Example 2.4.1. *This example illustrates the modalities for distribution of submatrices for processes according to "two-dimensional block-cyclic data layout scheme" algorithm using the BLACS functions.*

```
#include <mpi.h>
      #include <iostream>
      #include <iomanip>
      #include <string>
      #include <fstream>
      #include <sstream>
      using namespace std;
      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_pcoord(int, int, int*, int*);
          void Cblacs_gridexit(int);
          void Cblacs_barrier(int, const char*);
```

```
          void Cdgerv2d(int, int, int, double*, int, int, int);
          void Cdgesd2d(int, int, int, double*, int, int, int);
          int numroc_(int*, int*, int*, int*, int*);
      }
      int main(int argc, char **argv)
      {
// Initialization
          int mpirank,namelen;
          char processor_name[MPI_MAX_PROCESSOR_NAME];
          MPI_Init(&argc, &argv);
          MPI_Get_processor_name(processor_name,&namelen);
          MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
          bool mpiroot = (mpirank == 0);
          /* Helping vars */
          int iZERO = 0;
          if (argc < 6) {
              if (mpiroot)
                  cerr << "Usage: matrixTest matrixfile N M Nb Mb" << endl;
              MPI_Finalize();
              return 1;
          }
          int N, M, Nb, Mb;
          double *A_glob = NULL, *A_glob2 = NULL, *A_loc = NULL;
// Reeading data by root procces
          /* Parse command line arguments */
          if (mpiroot)
            {
              /* Read command line arguments */
              stringstream stream;
              stream << argv[2] << " " << argv[3] << " " << argv[4] << " " <<
                        argv[5];
              stream >> N >> M >> Nb >> Mb;
              /* Reserve space and read matrix (with transposition!) */
              A_glob  = new double[N*M];
              A_glob2 = new double[N*M];
              string fname(argv[1]);
              ifstream file(fname.c_str());
              for (int r = 0; r < N; ++r)
                {
                  for (int c = 0; c < M; ++c) {
                      file >> *(A_glob + N*c + r);
                  }
              }
              /* Print matrix */
              cout << "Matrix A:\n";
              for (int r = 0; r < N; ++r) {
                  for (int c = 0; c < M; ++c) {
                      cout << setw(3) << *(A_glob + N*c + r) << " ";
                  }
                  cout << "\n";
              }
              cout << endl;
          }
// BLACS initialization (a network of 2x3 processes are created)
          /* Begin Cblas context */
          /* We assume that we have 6 processes and place them in a 2-by-3 grid */
          int ctxt, myid, myrow, mycol, numproc;
          int procrows = 2, proccols = 3;
          Cblacs_pinfo(&myid, &numproc);
          Cblacs_get(0, 0, &ctxt);
          Cblacs_gridinit(&ctxt, "Row-major", procrows, proccols);
          Cblacs_pcoord(ctxt, myid, &myrow, &mycol);
          /* Print grid pattern */
          if (myid == 0)
              cout << "Processes grid pattern:" << endl;
          for (int r = 0; r < procrows; ++r)
```

```
          {
              for (int c = 0; c < proccols; ++c)
              {
                  Cblacs_barrier(ctxt, "All");
                  if (myrow == r && mycol == c)
                 {
printf("At grid position (%d,%d) is processor %d on the node ""%s""
\n",myrow,mycol,myid, processor_name);
                  }
              }
              Cblacs_barrier(ctxt, "All");
                 }
// Sharing the input data
          /* Broadcast of the matrix dimensions */
          int dimensions[4];
          if (mpiroot)
          {
              dimensions[0] = N;
              dimensions[1] = M;
              dimensions[2] = Nb;
              dimensions[3] = Mb;
          }
          MPI_Bcast(dimensions, 4, MPI_INT, 0, MPI_COMM_WORLD);
          N = dimensions[0];
          M = dimensions[1];
          Nb = dimensions[2];
          Mb = dimensions[3];
          /* Reserve space for local matrices */
          // Number of rows and cols owned by the current process
          int nrows = numroc_(&N, &Nb, &myrow, &iZERO, &procrows);
          int ncols = numroc_(&M, &Mb, &mycol, &iZERO, &proccols);
printf("The process (%d,%d) owns the %d rows  and %d  cols
\n",myrow,mycol,nrows,ncols);
          for (int id = 0; id < numproc; ++id)
          {
              Cblacs_barrier(ctxt, "All");
          }
          A_loc = new double[nrows*ncols];
          for (int i = 0; i < nrows*ncols; ++i) *(A_loc+i)=0.;
// Scattering the matrix
          /* Scatter matrix */
/* In fact it is here where is programmed the two dimensional block cyclic time
layout schemes algorithm for distributing matrices!!
*/
          int sendr = 0, sendc = 0, recvr = 0, recvc = 0;
          for (int r = 0; r < N; r += Nb, sendr=(sendr+1)%procrows)
            {
              sendc = 0;
              // Number of rows to be sent
              // Is this the last row block?
              int nr = Nb;
              if (N-r < Nb)
                  nr = N-r;
              for (int c = 0; c < M; c += Mb, sendc=(sendc+1)%proccols)
                {
                  // Number of cols to be sent
                  // Is this the last col block?
                  int nc = Mb;
                  if (M-c < Mb)
                      nc = M-c;
                  if (mpiroot)
                    {
                      // Send a nr-by-nc submatrix to process (sendr, sendc)
                      Cdgesd2d(ctxt, nr, nc, A_glob+N*c+r, N, sendr, sendc);
                  }
                  if (myrow == sendr && mycol == sendc)
```

```
                    {
                        // Receive the same data
                        // The leading dimension of the local matrix is nrows!
                        Cdgerv2d(ctxt, nr, nc, A_loc+nrows*recvc+recvr, nrows, 0, 0);
                        recvc = (recvc+nc)%ncols;
                    }
                }
                if (myrow == sendr)
                    recvr = (recvr+nr)%nrows;
            }
            /* Print local matrices */
            for (int id = 0; id < numproc; ++id) {
                if (id == myid) {
                    cout << "A_loc on node " << myid << endl;
                    for (int r = 0; r < nrows; ++r) {
                        for (int c = 0; c < ncols; ++c)
                            cout << setw(3) << *(A_loc+nrows*c+r) << " ";
                        cout << endl;
                    }
                    cout << endl;
                }
                Cblacs_barrier(ctxt, "All");
            }
// Gathering the matrix
            /* Gather matrix */
            sendr = 0;
            for (int r = 0; r < N; r += Nb, sendr=(sendr+1)%procrows) {
                sendc = 0;
                // Number of rows to be sent
                // Is this the last row block?
                int nr = Nb;
                if (N-r < Nb)
                    nr = N-r;
                for (int c = 0; c < M; c += Mb, sendc=(sendc+1)%proccols) {
                    // Number of cols to be sent
                    // Is this the last col block?
                    int nc = Mb;
                    if (M-c < Mb)
                        nc = M-c;
                    if (myrow == sendr && mycol == sendc) {
                        // Send a nr-by-nc submatrix to process (sendr, sendc)
                        Cdgesd2d(ctxt, nr, nc, A_loc+nrows*recvc+recvr, nrows, 0, 0);
                        recvc = (recvc+nc)%ncols;
                    }
                    if (mpiroot) {
                        // Receive the same data
                        // The leading dimension of the local matrix is nrows!
                        Cdgerv2d(ctxt, nr, nc, A_glob2+N*c+r, N, sendr, sendc);
                    }
                }
                if (myrow == sendr)
                    recvr = (recvr+nr)%nrows;
            }
            /* Print test matrix */
            if (mpiroot) {
                cout << "Matrix A test:\n";
                for (int r = 0; r < N; ++r) {
                    for (int c = 0; c < M; ++c) {
                        cout << setw(3) << *(A_glob2+N*c+r) << " ";
                    }
                    cout << endl;
                }
            }
        //Cleaning
            /* Release resources */
            delete[] A_glob;
```

```
            delete[] A_glob2;
            delete[] A_loc;
            Cblacs_gridexit(ctxt);
            MPI_Finalize();
        }
```

The result of the execution of the program

```
[MI_gr_TPS1@hpc]$ mpiCC_ScL Example2.4.1.cpp -o Example2.4.1.exe
[MI_gr_TPS1@hpc]$ /opt/openmpi/bin/mpirun -n 6 -host compute-0-10,compute-0-12
Example2.4.1.exe Matrix1.dat 9 9 2 2

Matrix A:
  0  -7  -7   0  -6   5   6   7   1
 -6  -4   1  -2  -9   9   8   7   2
 -1   7  10  -3   1  -7  -7  -4   3
  2   4  -4   3  -9  -1   0   1   4
 -6  -2   7   4  -4  -9  -3  10   5
 -3  -2  -1   5  -2   1 -10  -3   6
 -7   6   0  -4   5   7  -8   3   7
  4   8  -3  -8  -3 -10  -8  -8   8
  1   2   3   4   5   6   7   8   9
Processes grid pattern:
At grid position (0,0) is processor 0 on the node compute-0-10.local
At grid position (0,1) is processor 1 on the node compute-0-12.local
At grid position (0,2) is processor 2 on the node compute-0-10.local
At grid position (1,0) is processor 3 on the node compute-0-12.local
At grid position (1,1) is processor 4 on the node compute-0-10.local
At grid position (1,2) is processor 5 on the node compute-0-12.local
The process (0,2) owns the 5 rows  and 2  cols
The process (1,1) owns the 4 rows  and 3  cols
The process (0,0) owns the 5 rows  and 4  cols
The process (0,1) owns the 5 rows  and 3  cols
The process (1,2) owns the 4 rows  and 2  cols
The process (1,0) owns the 4 rows  and 4  cols

A_loc on node 0
  0  -7   6   7
 -6  -4   8   7
 -6  -2  -3  10
 -3  -2 -10  -3
  1   2   7   8

A_loc on node 1
 -7   0   1
  1  -2   2
  7   4   5
 -1   5   6
  3   4   9

A_loc on node 2
 -6   5
 -9   9
 -4  -9
 -2   1
  5   6

A_loc on node 3
 -1   7  -7  -4
  2   4   0   1
 -7   6  -8   3
  4   8  -8  -8

A_loc on node 4
```

```
 10  -3   3
 -4   3   4
  0  -4   7
 -3  -8   8

A_loc on node 5
  1  -7
 -9  -1
  5   7
 -3 -10

Matrix A test:
  0  -7  -7   0  -6   5   6   7   1
 -6  -4   1  -2  -9   9   8   7   2
 -1   7  10  -3   1  -7  -7  -4   3
  2   4  -4   3  -9  -1   0   1   4
 -6  -2   7   4  -4  -9  -3  10   5
 -3  -2  -1   5  -2   1 -10  -3   6
 -7   6   0  -4   5   7  -8   3   7
  4   8  -3  -8  -3 -10  -8  -8   8
  1   2   3   4   5   6   7   8   9
```