2.13 Linear Least Squares Problems

The linear least squares (LLS) problem is:

$$\min_x \left\| b - Ax \right\|_2 \qquad (3.1)$$

where *A* is an *m*-by-*n* matrix, *b* is a given *m* element vector and *x* is the *n* element solution vector. In the most usual case, $m \geq n$ and *rank(A)=n,* so that *A* has full rank and in this case the solution to problem (3.1) is unique; the problem is also referred to as finding a least squares solution to an over determined system of linear equations. When $m < n$ and *rank(A)=m* , there are an infinite number of solutions *x* that exactly satisfy *b-Ax=0.* In this case it is often useful to find the unique solution *x* that minimizes $\left\| x \right\|_2$, and the problem is referred to as finding a minimum norm solution to an underdetermined system of linear equations. The driver routine PxGELS solves problem (3.1) on the assumption that *rank(A)<min(m,n)* (in other words, *A* has full rank) finding a least squares solution of an over determined system when $m > n$, and a minimum norm solution of an underdetermined system when $m < n$. PxGELS uses a QR or LQ factorization of *A* and also allows *A* to be replaced by A^T in the statement of the problem (or by A^H if A is complex). In the general case when we may have *rank(A)<min(m,n)* (in other words, *A* may be rank-deficient) we seek the minimum norm least squares solution *x* that minimizes both $\left\| x \right\|_2$ and $\left\| b - Ax \right\|_2$. The LLS driver routines are listed in table 3.3. All routines allow several right-hand-side vectors and corresponding solutions x to be handled in a single call, storing these vectors as columns of matrices *B* and *X*, respectively. Note, however, that equation (3.1) is solved for each right-hand-side vector independently; this is not the same as finding a matrix *X* that minimizes $\left\| b - Ax \right\|_2$.

Operation	Single Precision		Double Precizion	
	Real	Complex	Real	Complex
Solve LLS using QR or LQ factorization	PSGELS	PCGELS	PDGELS	PZGELS

Table 3.3

Same properties of orthogonal matrices.

Let *Q nxn* matrice. *Q* is called orthogonal if $Q^TQ=I$. It is easy to invert orthogonal matrix: $Q^{-1}=Q^T$. Example: $Q = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{pmatrix}$,

$Q^T Q = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. Orthogonal transformations preserve the dot product: $A\vec{x} \cdot Q\vec{y} = \vec{x} \cdot \vec{y}$. The composition of two orthogonal transformations is orthogonal. The inverse of orthogonal transformation is orthogonal.

Orthogonal Factorizations and Least-squares Problems

In linear algebra, a QR decomposition (also called a QR factorization) of a matrix is a decomposition of a matrix *A* into a product *A=QR* of an orthogonal matrix *Q* and an upper triangular matrix *R*. QR decomposition is often used to solve the linear least squares problem,

and is the basis for a particular eigenvalue algorithm, the QR algorithm. A real matrix Q is **orthogonal** if $Q^TQ = I$; a complex matrix Q is **unitary** if $Q^HQ = I$. Orthogonal or unitary matrices have the important property that they leave the two-norm of a vector invariant, so that $\|x\|_2 = \|Qx\|_2$. They help to maintain numerical stability because they do not amplify rounding errors.

The *QR* factorization can be used to solve the linear least-squares problem of (3.1) when $m \geq n$ and A is of full rank. We can always choose *m-n* more orthogonal vector $\tilde{Q}$ so that $[Q, \tilde{Q}]$ is a square orthogonal matrix. Then $\|Ax - b\|_2^2 = \left\|[Q, \tilde{Q}](Ax - b)\right\|_2^2 = \left\|\begin{bmatrix} Q^T \\ \tilde{Q}^T \end{bmatrix}(QRx - b)\right\|_2^2 =$

$$= \left\|\begin{bmatrix} Q^T(QRx - b) \\ \tilde{Q}^T(QRx - b) \end{bmatrix}\right\|_2^2 = \left\|\begin{bmatrix} I^{n\times n} \\ 0^{(m-n)\times n} \end{bmatrix}Rx - \begin{bmatrix} Q^Tb \\ \tilde{Q}^Tb \end{bmatrix}\right\|_2^2 = \left\|\begin{bmatrix} Rx - Q^Tb \\ -\tilde{Q}^Tb \end{bmatrix}\right\|_2^2 = \|Rx - Q^Tb\|_2^2 + \|\tilde{Q}^Tb\|_2^2 \geq \|\tilde{Q}^Tb\|_2^2.$$

We can solve $Rx-Q^Tb=0$ for *x*, since *A* and *R* have the same *rank=n*, and so *R* is non-singular. *Then* $x=R^{-1}Q^Tb$, and the $\min_x \|b - Ax\|_2$ is equal to $\|\tilde{Q}^Tb\|_2$.

The second, slightly different derivation that does not use the matrix $\tilde{Q}$ is the following. Rewrite $Ax-b$ as $Ax-b=QRx-b= QRx -(QQ^T+I-QQ^T)b=Q(Rx-Q^Tb)-(I-QQ^T)b$. Note that the vectors $Q(Rx-Q^Tb)$ and[1] $(I-QQ^T)b$ are orthogonal (two vectors are orthogonal or perpendicular if their dot product is zero), because $(Q(Rx-Q^Tb))^T((I-QQ^T)b)=(Rx-Q^Tb)^T[Q^T(I-QQ^T)]b=(Rx-Q^Tb)^T[0]b=0$. Therefore, by the Pythagorean theorem,

$\|Ax - b\|_2^2 = \|Q(Rx - Q^Tb)\|_2^2 + \|(I - QQ^T)b\|_2^2 = \|Rx - Q^Tb\|_2^2 + \|(I - QQ^T)b\|_2^2$, because

$\|Q(Rx - Q^Tb)\|_2^2 = \|(Rx - Q^Tb)\|_2^2$. This sum of squares is minimized when the first term is zero, i.e. $x=R^{-1}Q^Tb$.

Finally, here is the third derivation that starts from the normal equation solution:

$$x = (A^TA)^{-1}A^Tb = (R^TQ^TQR)^{-1}R^TQ^Tb = (R^TR)^{-1}R^TQ^Tb = R^{-1}R^{-T}R^TQ^Tb = R^{-1}Q^Tb.$$

or

$$Q\cdot R\cdot x= b \Rightarrow \mathrm{Q^T}\ (Q\cdot R\cdot x= b)\ \Rightarrow R\cdot x= Q^Tb \Rightarrow \mathrm{x=R^{-1}Q^Tb}.$$

Syntaxes of the routine for compute the Linear Least Squares Problems

```
CALL PDGELS|PZGELS (transa, m, n, nrhs, a, ia, ja, desc_a,
                    b, ib, jb, desc_b, work, lwork, info)
void pdgels_(char *trans, int *m, int *n, int *nrhs,
                    double *a, int *ia, int *ja, int *desca,
```

[1] $\mathrm{Q^TQ}$ is not equal to $\mathrm{QQ^T}$.

```
                double *b, int *ib, int *jb, int *descb,
                double *work, int *lwork, int *info);
```

Purpose: PDGELS solves over determined or underdetermined real linear systems involving a real general matrix *A* or its transpose, using a QR or LQ factorization. It is assumed that *A* has full rank. The following options are provided:
If transa='N' and m≥n: find the least squares solution of an over determined system; that is, solve the least squares problem: minimize $\|b - Ax\|_2$
If transa = 'N' and m < n: find the minimum norm solution of an underdetermined system; that is, the problem is: Ax=b
For PDGELS:
If transa = 'T' and m ≥ n: find the minimum norm solution of an underdetermined system; that is, the problem is $A^Tx=b$
If transa = 'T' and m < n: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\|b - Ax\|_2$
For PZGELS:
If transa = 'C' and m ≥ n: find the minimum norm solution of an underdetermined system; that is, the problem is $A^Hx=b$
If transa = 'C' and m < n: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\|b - A^H x\|_2$
In the formulas above:
A represents the global general submatrix *A[ia:ia+m-1, ja:ja+n-1]*
If transa = 'N':
B represents the global general submatrix *B[ib:ib+m-1, jb:jb+nrhs-1]* containing the right-hand sides in its columns.
X represents the global general submatrix B[ib:ib+n-1, jb:jb+nrhs-1] containing the solution vectors in its columns.
If transa ≠ 'N':
B represents the global general submatrix B[ib:ib+n-1, jb:jb+nrhs-1] containing the right-hand sides in its columns.
X represents the global general submatrix B[ib:ib+m-1, jb:jb+nrhs-1] containing the solution vectors in its columns.

Input Parameters

`transa`

indicates the form of matrix *A* used in the system of equations, where:
If transa = 'N', matrix *A* is used.
If transa = 'T', matrix *A*T is used.
If transa = 'C', matrix *A*H is used.
Scope: global
Specified as: a single character; transa = 'N', 'T', or 'C'.

`m`

is the number of rows in submatrix *A* used in the computation.
Scope: global
Specified as: a fullword integer; m ≥ 0.

`n`

is the number of columns in submatrix *A* used in the computation.
Scope: **global**
Specified as: a fullword integer; n ≥ 0.

nrhs

is the number of right-hand sides; that is the number of columns in submatricies used in the computation.
Scope: **global**
Specified as: a fullword integer; nrhs ≥ 0.

a

is the local part of the global general matrix *A*. This identifies the **first element** of the local array A. This subroutine computes the location of the first element of the local subarray used, based on ia, ja, desc_a, p, q, myrow, and mycol; therefore, the leading LOCp(ia+m-1) by LOCq(ja+n-1) part of the local array A must contain the local pieces of the leading ia+m-1 by ja+n-1 part of the global matrix.
Note: No data should be moved to form *A*T or *A*H; that is, the matrix *A* should always be stored in its untransposed form.
Scope: **local**
Specified as: an LLD_A by (at least) LOCq(N_A) array, containing numbers of the data type indicated in Table 1. Details about the block-cyclic data distribution of global matrix *A* are stored in desc_a.

ia

is the row index of the global matrix *A*, identifying the first row of the submatrix *A*.
Scope: **global**
Specified as: a fullword integer; 1 ≤ ia ≤ M_A and ia+m-1 ≤ M_A.

ja

is the column index of the global matrix *A*, identifying the first column of the submatrix *A*.
Scope: **global**
Specified as: a fullword integer; 1 ≤ ja ≤ N_A and ja+n-1 ≤ N_A.

desc_a

is the array descriptor for global matrix *A*.

b

is the local part of the global general matrix B, containing the right-hand sides of the system. This identifies the first element of the local array B. This subroutine computes the location of the first element of the local subarray used, based on ib, jb, desc_b, p, q, myrow, and mycol.
If transa = 'N', the leading LOCp(ib+m-1) by LOCq(jb+nrhs-1) part of the local array B must contain the local pieces of the leading ib+m-1 by jb+nrhs-1 part of the global matrix; otherwise, the leading LOCp(ib+n-1) by LOCq(jb+nrhs-1) part of the local array B must contain the local pieces of the leading ib+n-1 by jb+nrhs-1 part of the global matrix.
Scope: **local**

ib

is the row index of the global matrix B, identifying the first row of the submatrix B.
Scope: **global**
Specified as: a fullword integer; 1 ≤ ib ≤ M_B and ib + max(m, n)-1 ≤ M_B.

jb

is the column index of the global matrix B, identifying the first column of the submatrix B.
Scope: **global**
Specified as: a fullword integer; 1 ≤ jb ≤ N_B and jb+nrhs-1 ≤ N_B.

`desc_b`
is the array descriptor for global matrix B.

`work`
has the following meaning:
If lwork = 0, work is ignored.
If lwork ≠ 0, work is the work area used by this subroutine, where:
If lwork ≠ -1, its size is (at least) of length lwork.
If lwork = -1, its size is (at least) of length 1.
Scope: local

`lwork`
is the number of elements in array WORK.
Scope:
If lwork ≥ 0, lwork is local
If lwork = -1, lwork is global
Specified as: a fullword integer; where:
If lwork = 0, PDGELS and PZGELS dynamically allocate the work area used by this subroutine. The work area is deallocated before control is returned to the calling program. This option is an extension to the ScaLAPACK standard.
If lwork = -1, PDGELS and PZGELS dynamically perform a work area query and returns the minimum size of work in work1. No computation is performed and the subroutine returns after error checking is complete.
Otherwise, it must have the following value:
lwork ≥ ltau + max(lwf, lws)
where:
If m ≥ n, then:
ltau = NUMROC(ja + min(m, n)-1, NB_A, mycol, CSRC_A, npcol)
lwf = NB_A (mpa0 + nqa0 + NB_A)
lws = max((NB_A (NB_A-1)) / 2, (nrhsqb0 + mpb0) NB_A) + (NB_A)(NB_A)
If m < n, then:
ltau = NUMROC(ia + min(m, n)-1, MB_A, myrow, RSRC_A, nprow)
lwf = MB_A (mpa0 + nqa0 + MB_A)
lws = max((MB_A (MB_A-1)) / 2, (npb0 + max(nqa0 + NUMROC(NUMROC(n + iroffb, MB_A, 0, 0, nprow), MB_A, 0, 0, lcmp), nrhsqb0)) MB_A) + (MB_A)(MB_A)
where:
lcm = ilcm(nprow, npcol)
lcmp = lcm/nprow
iroffa = mod(ia-1, MB_A)
icoffa = mod(ja-1, NB_A)
iarow = mod(RSRC_A + (ia-1)/MB_A, nprow)
iacol = mod(CSRC_A + (ja-1)/NB_A, npcol)
mpa0 = NUMROC(m+iroffa, MB_A, myrow, iarow, nprow)
nqa0 = NUMROC(n+icoffa, NB_A, mycol, iacol, npcol)
iroffb = mod(ib-1, MB_B)
icoffb = mod(jb-1, NB_B)
ibrow = mod(RSRC_B + (ib-1)/MB_B, nprow)
ibcol = mod(CSRC_B + (jb-1)/NB_B, npcol)
mpb0 = NUMROC(m+iroffb, MB_B, myrow, ibrow, nprow)
npb0 = NUMROC(n+iroffb, MB_B, myrow, ibrow, nprow)
nrhsqb0 = NUMROC(nrhs+icoffb, NB_B, mycol, ibcol, npcol)

`info`

On Return

a

is the updated local part of the global general matrix A. Matrix A is overwritten; the original input is not preserved.
Scope: **local**
Returned as: an LLD_A by (at least) LOCq(N_A) array, containing numbers of the data type indicated in Table 1. Details about the block-cyclic data distribution of global matrix A are stored in desc_a.

b

is the updated local part of the global general matrix B, overwritten by the solution vectors stored columnwise.
If transa = 'N' and m ≥ n, rows ib:ib+n-1 contain the least squares solution vectors. The residual sum of squares for each column is given by the sum of squares of elements ib+n:ib+m-1 in that column.
If transa = 'N' and m < n, rows ib:ib+n-1 contain the minimum norm solution vectors.
If transa ≠ 'N' and m ≥ n, rows ib:ib+m-1 contain the minimum norm solution vectors.
If transa ≠ 'N' and m < n, rows ib:ib+m-1 contain the least squares solution vectors. The residual sum of squares for each column is given by the sum of squares of elements ib+m:ib+n-1 in that column.
Scope: **local**
Returned as: an LLD_B by (at least) LOCq(N_B) array, containing numbers of the data type indicated in Table 1. Details about the block-cyclic data distribution of global matrix B are stored in desc_b.

work

is the work area used by this subroutine if lwork ≠ 0.
Scope: **local**
Returned as: an area of storage, where:
If lwork ≥ 1 or lwork = -1, then work1 is set to the minimum lwork value and contains numbers of the data type indicated in Table 1. Except for work1, the contents of work are overwritten on return.

info

indicates that a successful computation occurred.
Scope: **global**
Returned as: a fullword integer; info = 0.

Example 2.13.1. The using of function pdgels

```
#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 BB(i,j) BB[(i)*n+(j)]
#define A(i,j) A[(i)*n+(j)]
#define B(i,j) B[(i)*n+(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 exit( int error code);
void   Cblacs barrier(int, const char*);
int    numroc ( int *n, int *nb, int *iproc, int *isrcproc, int *nprocs);
void   descinit ( int *desc, int *m, int *n, int *mb, int *nb, int *irsrc,
int *icsrc,
                  int *ictxt, int *lld, int *info);
double pdlamch ( int *ictxt , char *cmach);
double pdlange ( char *norm, int *m, int *n, double *A, int *ia, int *ja, int
*desca, double *work);
void pdlacpy ( char *uplo, int *m, int *n, double *a, int *ia, int *ja, int
*desca,
            double *b, int *ib, int *jb, int *descb);
void pdgesv ( int *n, int *nrhs, double *A, int *ia, int *ja, int *desca,
int* ipiv,
            double *B, int *ib, int *jb, int *descb, int *info);
void pdgemm ( char *TRANSA, char *TRANSB, int * M, int * N, int * K, double *
ALPHA,
            double * A, int * IA, int * JA, int * DESCA, double * B, int *
IB, int * JB, int * DESCB,
            double * BETA, double * C, int * IC, int * JC, int * DESCC );
int  indxg2p ( int *indxglob, int *nb, int *iproc, int *isrcproc, int
*nprocs);
int  indxl2g (int*, int*, int*, int*, int*);
void pdgels (char *trans, int *m, int *n, int *nrhs, double *a, int *ia, int
*ja, int *desca, double *b, int *ib, int *jb, int *descb, double *work, int
*lwork, int *info);
void Cpdlaprnt(int *m, int *n, double *a, int *ia, int *ja, int *desca, int
*irprnt,
               int *icprnt, char *cmatnm, int *nout, double *work);
void Cpdlawrite( char *filnam, int *m, int *n, double *a, int *ia, int *ja,
int *desca,
                  int *irwrit, int *icwrit, double *work );
void Cpdlaread( char *filnam, double *a, int *desca, int *irread, int
*icread, double *work );
}

static int max( int a, int b ){
        if (a>b) return(a); else return(b);
}

/* Parameters */
#define M 6
#define N 4
#define NRHS 1
#define LDA M
#define LDB M

/* Main program */
int main(int argc, char **argv) { //#0

        /* Locals */
        int m = M, n = N, nrhs = NRHS, lda = LDA, ldb = LDB, info, lwork;
        //double wkopt;
```

```
        double *work;
int iam, nprocs;
int myrank mpi, nprocs mpi;
int ictxt, nprow, npcol, myrow, mycol;
int descA[9], descB[9];
double *AA,*BB;
double *A, *B;
int iloc,jloc,i,j,k;
int itemp;
int mb,nb; // blok dimension
int izero=0,ione=1;
double MPIt1, MPIt2, MPIelapsed;
int mpirank,namelen;
char processor name[MPI MAX PROCESSOR NAME];
MPI Init( &argc, &argv);
MPI Comm rank(MPI COMM WORLD, &myrank mpi);
MPI Comm size(MPI COMM WORLD, &nprocs mpi);
MPI Get processor name(processor name,&namelen);
bool mpiroot = (myrank mpi == 0);
       /*
          if (argc < 6) {
              if (mpiroot)
                  cerr << "Usage: HB pdgels.exe Matrix pdgels.dat N M Nb Mb"
<< endl;
              MPI Finalize();
              return 1;
          }*/
nprow = 2; npcol = 2;
mb=2; nb = 2;
if (nprow*npcol>nprocs mpi){//#1
if (myrank mpi==0)
printf(" **** ERROR : we do not have enough processes available to make a p-
by-q process grid ***\n");
printf(" **** Bye-bye
***\n");
MPI Finalize(); exit(1);
                  }//#1
Cblacs pinfo( &iam, &nprocs ) ;
Cblacs get( -1, 0, &ictxt );
Cblacs gridinit( &ictxt, "Row", nprow, npcol );
Cblacs gridinfo( ictxt, &nprow, &npcol, &myrow, &mycol );
AA = (double*)malloc(m*n*sizeof(double));// matricea "globala" pentru AA*X=B
BB = (double*)malloc(m*sizeof(double));//vectorul "global" pentru AA*X=B
 fstream f;
            f.open("data.dat",fstream::in);
            f >> m >> n >> nrhs;
if (myrank mpi==0)
printf("Numarul de randuri= %d, numarul de coloane=%d, numarul de vectori din
partea dreapta= %d\n",m,n,nrhs);
            AA = new double[m*n];
            BB = new double[m*nrhs];
            for(j = 0; j< m; ++j)
            {
                  for(k =0; k < n; ++k)
                              f >> *(AA + j*n + k);
                  }
            for(j = 0; j< m; ++j)
            {
```

```
                  for(k =0; k < nrhs; ++k)
                        f >> *(BB + j*nrhs + k);
            }
int mA = numroc ( &m, &mb, &myrow, &izero, &nprow );
int nA = numroc ( &n, &nb, &mycol, &izero, &npcol );
int mB = numroc ( &m, &mb, &myrow, &izero, &nprow );
int nqrhs = numroc ( &nrhs, &nb, &mycol, &izero, &npcol );
if (mpiroot)
{//#2
 /* Print matrix */
              cout << "Matrix AA:\n";
//Cblacs barrier(ictxt, "All");
              for (i = 0; i < m; ++i) {
                  for (j = 0; j < n; ++j) {
                      cout << setw(5) << *(AA + i*n + j) << " ";
                                        }
                  cout << "\n";
                                    }
              cout << endl;
cout << "Matrix BB:\n";
//Cblacs barrier(ictxt, "All");
              for (i = 0; i < m; ++i) {
                  for (j = 0; j < nrhs; ++j) {
                      cout << setw(5) << *(BB +i*nrhs+j) << " ";
                                        }
                  cout << "\n";
                                    }
              cout << endl;
}//#2
Cblacs barrier(ictxt, "All");
A = new double[mA*nA];
B = new double[mB*nqrhs];
//=== se complecteaza cu valori matricele locale
//if ((myrow==0)&(mycol==1))
{
  for(iloc=0;iloc<mA;iloc++)
  for(jloc=0;jloc<nA;jloc++){
  int fortidl = iloc + 1;
  int fortjdl = jloc + 1;
  i = indxl2g (&fortidl, &mb, &myrow, &izero, &nprow)-1;
  j = indxl2g (&fortjdl, &nb, &mycol, &izero, &npcol)-1;
  A[iloc*nA+jloc]=AA(i,j);
                             }

  for(iloc=0;iloc<mB;iloc++)
   for (jloc = 0; j < nqrhs; jloc++) {
  int fortidl = iloc + 1;
  int fortjdl = jloc + 1;
  i = indxl2g (&fortidl, &mb, &myrow, &izero, &nprow)-1;
  j = indxl2g (&fortjdl, &nb, &mycol, &izero, &npcol)-1;
  //               B[iloc*mB+jloc]=BB(i,j);
  //i = indxl2g (&iloc, &mb, &myrow, &izero, &nprow);
  //j = indxl2g (&jloc, &nb, &mycol, &izero, &npcol);
                //B[iloc*mB+jloc]=BB(i,j);
   B[iloc*nqrhs+jloc]=BB(i,j);             }
}

// === se tiparesc matricele locale
```

```
Cblacs barrier(ictxt, "All");
if ((myrow==0)&(mycol==0))
{
printf("For (%d,%d) process local matrix A is\n",myrow,mycol);
for (i = 0; i < mA; ++i)
                    {
  for (j = 0; j < nA; ++j)
  cout << setw(5) << *(A+nA*i+j) << "   ";
  cout << endl;
                    }
}
Cblacs barrier(ictxt, "All");
if (mycol==0)
{
printf("For process (%2d%2d) the local vector B is \n",myrow,mycol);
for (i = 0; i < mB; ++i)
                    {
 for (j = 0; j < nqrhs; ++j)
cout << setw(3) << *(B+i*nqrhs+j) << "   ";
cout << endl;
                    }
}

Cblacs barrier(ictxt, "All");
free(AA);
free(BB);
itemp = max(1,mA);
descinit (descA, &m, &n, &mb, &nb, &izero, &izero, &ictxt, &itemp, &info);
descinit (descB, &m, &nrhs, &mb, &nb, &izero, &izero, &ictxt, &itemp, &info);
//  get correct size of work array
work = (double*)malloc(2*sizeof(double));//(double *)calloc(2,sizeof(double))
;
if (work==NULL){ printf("error of memory allocation for work on proc %dx%d
(1st time)\n",myrow,mycol); exit(0); }
                    lwork=-1;
pdgels ("N", &m, &n, &nrhs, A, &ione, &ione, descA, B, &ione, &ione, descB,
work, &lwork, &info);
lwork= (int)work[0];
                    free(work);
 //  real calculation
                    work = (double*)malloc(lwork*sizeof(double));//(double
*)calloc(lwork,sizeof(double)) ;
                    if (work==NULL){ printf("error of memory allocation work
on proc %dx%d\n",myrow,mycol); exit(0); }
pdgels ("N", &m, &n, &nrhs, A, &ione, &ione, descA, B, &ione, &ione, descB,
work, &lwork, &info);
Cblacs barrier(ictxt, "All");
if(mycol == 0 ){
printf("For process (%2d%2d) the solution is  %8.4f %8.4f %8.4f
%8.4f\n",myrow,mycol,B[0],B[1],B[2],B[3]);
      }
/*
if (myrow==0 & mycol==0)
{
printf("For process (%2d%2d) the solution is \n",myrow,mycol);
for (i = 0; i < mB; ++i)
                    {
  //for (j = 0; j < nqrhs; ++j)
```

```
cout << setw(3) << *(B+i) << "   ";
cout << endl;
                  }
}
*/
Cblacs barrier(ictxt, "All");
free(A);
free(B);
free(work);
Cblacs gridexit(ictxt);
//Cblacs exit(0);
MPI Finalize();
exit(0);
}//#0
```