RANDUTV-DM

1.0

Generated by Doxygen 1.8.17

1 Introduction 1

1.1 Basic installation steps for DM . . . 1

2 File Index 3

2.1 File List . . . 3

3 File Documentation 5

3.1 pdgeutv.f90 File Reference . . . 5

3.1.1 Function/Subroutine Documentation . . . 5

3.1.1.1 compute_local_svd() . . . 6

3.1.1.2 generate_normal_matrix() . . . 7

3.1.1.3 generate_normal_random_number() . . . 8

3.1.1.4 multiply_bab() . . . 9

3.1.1.5 multiply_bba() . . . 11

3.1.1.6 pdgeutv() . . . 13

3.1.1.7 pdmygeqr2() . . . 22

3.1.1.8 pdmylarfb_fc() . . . 24

3.2 pdttrandutv.f90 File Reference . . . 26

3.2.1 Function/Subroutine Documentation . . . 27

3.2.1.1 check_utv_resids() . . . 27

3.2.1.2 compar_svd() . . . 30

3.2.1.3 generate_random_matrix() . . . 33

3.2.1.4 pd_test_and_time_randutv() . . . 34

3.2.1.5 pdttrandutv() . . . 37

3.2.1.6 print_distributed_matrix() . . . 38

3.2.1.7 print_local_matrix() . . . 39

3.3 README.md File Reference . . . 40

Index 41

Chapter 1

Introduction

`randUTV SM-DM` is a software package that accompanies the submission *Efficient algorithms for computing a rank-revealing UTV factorization on parallel computing architectures* and includes an algorithm-by-blocks implementation for shared-memory architectures (`SM` from now on) and a distributed memory implementation (`DM` from now on) to compute a *full* factorization of a given matrix that provides low-rank approximations with near-optimal error.

This guide complements the submitted paper and includes basic setup and execution steps for the package. The provided software package includes all the algorithms described in the paper, namely:

- Directory `basic_randutv_sm` contains the *algorithm-by-blocks* for shared-memory architectures that leverages the `libflame` SuperMatrix infrastructure for the implementation.
- Directory `basic_randutv_dm` contains the distributed-memory implementation based on ScaLAPACK.

1.1 Basic installation steps for DM

The steps to install this subpackage are the following:

1. Uncompress the file.
2. Edit the first part of the "Makefile" file.

   The user must define the following variables (maybe replace the current values assigned to these variables):

   - the Fortran 90 compiler,
   - the Fortran 90 compiler flags,
   - the Fortran 90 loader or linker (usually the same as the compiler),
   - the Fortran 90 loader flags, and
   - the libraries to be employed:
     - the ScaLAPACK library (and the PBLAS and BLACS libraries, if not included inside it),
     - the LAPACK library, and
     - the BLAS library.

   There is no need to define the variable MKLROOT (and install the MKL library) if public-domain ScaLAPACK, LAPACK and BLAS libraries are employed. No more changes are required in this file.
3. Type "make" to compile and generate the executable file.
4. Execute the driver with one of the following commands: mpirun -np 4 pdttrandutv.x mpiexec -n 4 pdttrandutv.x

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

pdgeutv.f90 5
pdttrandutv.f90 26

Chapter 3

File Documentation

3.1 pdgeutv.f90 File Reference

Functions/Subroutines

- subroutine pdgeutv (m, n, a, desca, build_u, u, descu, build_v, v, descv, q, info)

  Computes a randUTV factorization A = U ∗ T ∗ V'.
- subroutine pdmygeqr2 (m, n, a, ia, ja, desca, tau)

  Computes the QR factorization of a real distributed m-by-n matrix sub(A) = A(IA:IA+M-1,JA:JA+N-1) = Q ∗ R.
- subroutine pdmylarfb_fc (side, trans, m, n, k, v, iv, jv, descv, t, c, ic, jc, descc)

  Applies a real block reflector Q or its transpose Q∗∗T to a real distributed M-by-N matrix sub(C) = C(IC:IC+M-1,JC↩ :JC+N-1) from the left or from the right. Matrix Q is intrisically stored in matrix v (Householder vectors) and vector t (tau factors).
- subroutine compute_local_svd (m, n, a, lda, u, ldu, s, vt, ldvt, info)

  Computes the SVD factorization of local matrix A, and returns orthonormal matrices U and VT, and the singular values in s. This operation is performed on local matrices and vectors.
- subroutine multiply_bba (transa, transb, m, n, a, desca, b, ib, jb, descb)

  Computes the following operation: B := B ∗ A, where A and B are transposed according to arguments "transa" and "transb". Matrix B is m x n.
- subroutine multiply_bab (transa, transb, m, n, a, desca, b, ib, jb, descb)

  Computes the following operation: B := A ∗ B, where A and B are transposed according to arguments "transa" and "transb". Matrix B is m x n.
- subroutine generate_normal_matrix (m, n, a, ia, ja, desca)

  Generates a distributed random matrix with elements in a normal distribuition.
- double precision function generate_normal_random_number (mu, sigma)

3.1.1 Function/Subroutine Documentation

3.1.1.1 compute_local_svd()

```
subroutine compute_local_svd (
             integer m,
             integer n,
             double precision, dimension( lda, * ) a,
             integer lda,
             double precision, dimension( ldu, * ) u,
             integer ldu,
             double precision, dimension( * ) s,
             double precision, dimension( ldvt, * ) vt,
             integer ldvt,
             integer info )
```

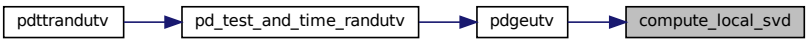
Computes the SVD factorization of local matrix A, and returns
orthonormal matrices U and VT, and the singular values in s.
This operation is performed on local matrices and vectors.

Definition at line 1029 of file pdgeutv.f90.

```
1029   !
1030   implicit none
1031   !
1032   ! .. Scalar Arguments ..
1033   integer            m, n, lda, ldu, ldvt, info
1034   ! ..
1035   ! .. Array Arguments ..
1036   double precision   a( lda, * ), u( ldu, * ), s( * ), vt( ldvt, * )
1037   ! ..
1038   !
1039   !  Purpose
1040   !  =======
1041   !
1042   !  It computes the SVD factorization of local matrix A, and returns
1043   !  orthonormal matrices U and VT, and the singular values in s.
1044   !  This operation is performed on local matrices and vectors.
1045   !
1046   !  =====================================================================
1047   !
1048   ! .. Local Scalars ..
1049   integer            len_work, allocstat
1050   double precision   scalar_work
1051   ! ..
1052   ! .. Local Arrays ..
1053   double precision, dimension ( : ), allocatable :: work
1054   ! ..
1055   ! .. External subroutines ..
1056   external           dgesvd
1057   ! ..
1058   ! .. Intrinsic Functions ..
1059   intrinsic          int
1060   ! ..
1061   ! .. Executable Statements ..
1062
1063   !
1064   ! Obtain the optimal real workspace length.
1065   !
1066   len_work = -1
1067   call dgesvd( 'All', 'All', m, n, a, lda, s, u, ldu, vt, ldvt, &
1068                scalar_work, len_work, info )
1069   if( info /= 0 ) then
1070     write ( *, * ) '*** ERROR in compute_local_svd: ', &
1071         'Info of call to compute wk length: ', info
1072   end if
1073   len_work = int( scalar_work )
1074
1075   !
1076   ! Allocate workspace.
1077   !
1078   allocate( work( len_work ), stat = allocstat )
1079   if ( allocstat /= 0 ) stop 'compute_local_svd: *** Not enough memory for work ***'
1080
1081   !
1082   ! Compute singular values and vectors.
1083   !
1084   call dgesvd( 'All', 'All', m, n, a, lda, s, u, ldu, vt, ldvt, &
1085                work, len_work, info )
1086   if( info /= 0 ) then
```

```
1087      write ( *, * ) '*** ERROR in compute_local_svd: Info of dgesvd: ', info
1088    end if
1089
1090    !
1091    ! Deallocate workspace.
1092    !
1093    deallocate( work )
1094
1095    !
1096    ! End of compute_local_svd
1097    !
1098    return
```

Here is the caller graph for this function:



3.1.1.2 generate_normal_matrix()

```
subroutine generate_normal_matrix (
             integer m,
             integer n,
             double precision, dimension( * ) a,
             integer ia,
             integer ja,
             integer, dimension( * ) desca )
```

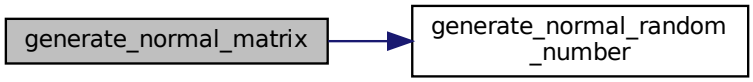
Generates a distributed random matrix with elements in a normal distribuition.

Definition at line 1299 of file pdgeutv.f90.

```
1299    !
1300    implicit none
1301    !
1302    ! .. Scalar Arguments ..
1303    integer            m, n, ia, ja
1304    ! ..
1305    ! .. Array Arguments ..
1306    integer            desca( * )
1307    double precision   a( * )
1308    ! ..
1309    !
1310    !  Purpose
1311    !  =======
1312    !
1313    !  It generates a distributed random matrix with elements in a normal
1314    !  distribuition.
1315    !
1316    !  ====================================================================
1317    !
1318    ! .. Parameters ..
1319    integer            BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DTYPE_, &
1320                       LLD_, MB_, M_, NB_, N_, RSRC_
1321    parameter( block_cyclic_2d = 1, dlen_ = 9, dtype_ = 1, &
1322                         ctxt_ = 2, m_ = 3, n_ = 4, mb_ = 5, nb_ = 6, &
1323                         rsrc_ = 7, csrc_ = 8, lld_ = 9 )
1324    double precision   ONE, ZERO
1325    parameter( one = 1.0d+0, zero = 0.0d+0 )
1326    ! ..
1327    ! .. Local Scalars ..
1328    integer            ictxt, nprow, npcol, myrow, mycol, lda, nb, &
1329                       i, j, mp, nq, ip, jq, iarow, jacol
1330    ! ..
1331    ! .. External Functions ..
1332    integer            numroc
```

```
1333   double precision   generate_normal_random_number
1334   external           numroc, generate_normal_random_number
1335   ! ..
1336   ! .. External subroutines ..
1337   external           blacs_gridinfo, infog2l
1338   ! ..
1339   ! .. Executable Statements ..
1340
1341   !
1342   ! Get grid parameters.
1343   !
1344   ictxt = desca( ctxt_ )
1345   call blacs_gridinfo( ictxt, nprow, npcol, myrow, mycol )
1346   lda   = desca( lld_ )
1347   nb    = desca( nb_ )
1348
1349   !
1350   ! Compute local indices.
1351   !
1352   call infog2l( ia, ja, desca, nprow, npcol, myrow, mycol, &
1353                 ip, jq, iarow, jacol )
1354   mp = numroc( ia + m - 1, nb, myrow, desca( rsrc_ ), nprow )
1355   nq = numroc( ja + n - 1, nb, mycol, desca( csrc_ ), npcol )
1356
1357   !
1358   ! Process elements in local matrix.
1359   !
1360
1361   do j = jq, nq
1362     do i = ip, mp
1363       !!!! num = generate_normal_random_number( ZERO, ONE )
1364       !!!! print *, myrow, mycol, i, j, num
1365       !!!! a( i + ( j - 1 ) * lda ) = num
1366       a( i + ( j - 1 ) * lda ) = generate_normal_random_number( zero, one )
1367     end do
1368   end do
1369
1370   !
1371   ! End of generate_normal_matrix
1372   !
1373   return
```

Here is the call graph for this function:



Here is the caller graph for this function:

3.1.1.3 generate_normal_random_number()

```
double precision function generate_normal_random_number (
            double precision mu,
            double precision sigma )
```

Definition at line 1378 of file pdgeutv.f90.

```
1378   !
1379   implicit none
1380   !
1381   ! .. Scalar Arguments ..
1382   double precision   mu, sigma
1383   ! ..
1384   !
1385   !  Purpose
1386   !  =======
1387   !
1388   !  It returns a double-precision random number with a normal distribution.
1389   !
1390   !  =====================================================================
1391   !
1392   ! .. Parameters ..
1393   double precision   ONE, ZERO
1394   parameter( one = 1.0d+0, zero = 0.0d+0 )
1395   ! ..
1396   ! .. Local Scalars ..
1397   logical, save ::          alternate = .false.
1398   double precision, save :: b1, b2
1399   double precision ::       u, c1, c2, a, factor
1400   ! ..
1401   ! .. Intrinsic Functions ..
1402   intrinsic                 log, sqrt, random_number
1403   ! ..
1404   ! .. Executable Statements ..
1405
1406   !
1407   ! Quick return.
1408   !
1409   if( alternate ) then
1410     alternate = .not. alternate
1411     generate_normal_random_number = mu + sigma * b2
1412     return
1413   end if
1414
1415   !
1416   ! Main loop.
1417   !
1418   do
1419     call random_number( u )
1420     c1 = -1.0 + 2.0 * u
1421     call random_number( u )
1422     c2 = -1.0 + 2.0 * u
1423     a  = c1 * c1 + c2 * c2;
1424     if( .not.( ( a == zero ).or.( a >= one ) ) ) exit
1425   end do
1426   factor    = sqrt( ( -2.0 * log( a ) ) / a );
1427   b1        = c1 * factor;
1428   b2        = c2 * factor;
1429   alternate = .not. alternate
1430   generate_normal_random_number = mu + sigma * b1
1431
1432   !
1433   ! End of generate_normal_random_number
1434   !
1435   return
```

Here is the caller graph for this function:

3.1.1.4 multiply_bab()

```
subroutine multiply_bab (
            character*( * ) transa,
            character*( * ) transb,
```

```
        integer m,
        integer n,
        double precision, dimension( * ) a,
        integer, dimension( * ) desca,
        double precision, dimension( * ) b,
        integer ib,
        integer jb,
        integer, dimension( * ) descb )
```

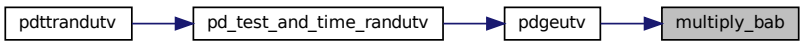
Computes the following operation: B := A ∗ B, where A and B are transposed according to arguments "transa" and "transb". Matrix B is m x n.

Definition at line 1199 of file pdgeutv.f90.

```
1199   !
1200   implicit none
1201   !
1202   ! .. Scalar Arguments ..
1203   character*( * )    transa, transb
1204   integer            m, n, ib, jb
1205   ! ..
1206   ! .. Array Arguments ..
1207   integer            desca( * ), descb( * )
1208   double precision   a( * ), b( * )
1209   ! ..
1210   !
1211   !  Purpose
1212   !  =======
1213   !
1214   !  It computes the following operation:  B := A * B, where A and B are
1215   !  transposed according to arguments "transa" and "transb".
1216   !  Matrix B is m x n.
1217   !
1218   !  =====================================================================
1219   !
1220   ! .. Parameters ..
1221   integer            BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DTYPE_, &
1222                      LLD_, MB_, M_, NB_, N_, RSRC_
1223   parameter( block_cyclic_2d = 1, dlen_ = 9, dtype_ = 1, &
1224                        ctxt_ = 2, m_ = 3, n_ = 4, mb_ = 5, nb_ = 6, &
1225                        rsrc_ = 7, csrc_ = 8, lld_ = 9 )
1226   double precision   ONE, ZERO
1227   parameter( one = 1.0d+0, zero = 0.0d+0 )
1228   ! ..
1229   ! .. Local Scalars ..
1230   integer            ictxt, nprow, npcol, myrow, mycol, nb, &
1231                      proc_row_of_ib_b, proc_col_of_jb_b, &
1232                      mpbc, nqbc, len_bc, allocstat
1233   ! ..
1234   ! .. Local Arrays ..
1235   integer            descbc( DLEN_ )
1236   double precision, dimension ( : ), allocatable :: bc
1237   ! ..
1238   ! .. External Functions ..
1239   integer            indxg2p, numroc
1240   external           indxg2p, numroc
1241   ! ..
1242   ! .. External Subroutines ..
1243   external           blacs_gridinfo, descset, pdgemm, pdlacpy
1244   ! ..
1245   ! .. Executable Statements ..
1246
1247   !
1248   ! Get grid parameters.
1249   !
1250   ictxt = descb( ctxt_ )
1251   call blacs_gridinfo( ictxt, nprow, npcol, myrow, mycol )
1252   nb    = descb( nb_ )
1253
1254   !
1255   ! Prepare descriptor for copy of matrix b.
1256   !
1257   proc_row_of_ib_b = indxg2p( ib, descb( mb_ ), myrow, descb( rsrc_ ), nprow )
1258   proc_col_of_jb_b = indxg2p( jb, descb( nb_ ), mycol, descb( csrc_ ), npcol )
1259   mpbc = numroc( m, nb, myrow, proc_row_of_ib_b, nprow )
1260   nqbc = numroc( n, nb, mycol, proc_col_of_jb_b, npcol )
1261   call descset( descbc, m, n, nb, nb, &
1262                 proc_row_of_ib_b, proc_col_of_jb_b, ictxt, max( 1, mpbc ) )
1263
1264   !
1265   ! Allocate copy of matrix b.
1266   !
```

```
1267   len_bc = max( 1, mpbc * nqbc )
1268   allocate( bc( len_bc ), stat = allocstat )
1269   if ( allocstat /= 0 ) stop 'Multiply_BAB: *** Not enough memory for bc ***'
1270
1271   !
1272   ! Copy b into bc.
1273   !
1274   call pdlacpy( 'All', m, n, b, ib, jb, descb, &
1275                              bc, 1, 1, descbc )
1276
1277   !
1278   ! Compute:  b := a * bc.
1279   !
1280   call pdgemm( transa, transb, m, n, m, &
1281                one, a, 1, 1, desca, bc, 1, 1, descbc, &
1282                zero, b, ib, jb, descb )
1283
1284   !
1285   ! Deallocate copy of matrix B.
1286   !
1287   deallocate( bc )
1288
1289   return
1290   !
1291   ! End of Multiply_BAB
1292   !
```

Here is the caller graph for this function:



3.1.1.5 multiply_bba()

```
subroutine multiply_bba (
             character*( * ) transa,
             character*( * ) transb,
             integer m,
             integer n,
             double precision, dimension( * ) a,
             integer, dimension( * ) desca,
             double precision, dimension( * ) b,
             integer ib,
             integer jb,
             integer, dimension( * ) descb )
```

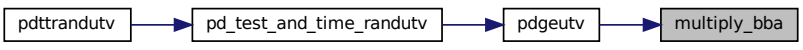
Computes the following operation: B := B ∗ A, where A and B are transposed according to arguments "transa" and "transb". Matrix B is m x n.

Definition at line 1107 of file pdgeutv.f90.

```
1107   !
1108   implicit none
1109   !
1110   ! .. Scalar Arguments ..
1111   character*( * )    transa, transb
1112   integer            m, n, ib, jb
1113   ! ..
1114   ! .. Array Arguments ..
1115   integer            desca( * ), descb( * )
1116   double precision   a( * ), b( * )
1117   ! ..
1118   !
```

```
1119  !  Purpose
1120  !  =======
1121  !
1122  !  It computes the following operation:  B := B * A, where A and B are
1123  !  transposed according to arguments "transa" and "transb".
1124  !  Matrix B is m x n.
1125  !
1126  !  =====================================================================
1127  !
1128  ! .. Parameters ..
1129  integer            BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DTYPE_, &
1130                     LLD_, MB_, M_, NB_, N_, RSRC_
1131  parameter( block_cyclic_2d = 1, dlen_ = 9, dtype_ = 1, &
1132                       ctxt_ = 2, m_ = 3, n_ = 4, mb_ = 5, nb_ = 6, &
1133                       rsrc_ = 7, csrc_ = 8, lld_ = 9 )
1134  double precision   ONE, ZERO
1135  parameter( one = 1.0d+0, zero = 0.0d+0 )
1136  ! ..
1137  ! .. Local Scalars ..
1138  integer            ictxt, nprow, npcol, myrow, mycol, nb, &
1139                     proc_row_of_ib_b, proc_col_of_jb_b, &
1140                     mpbc, nqbc, len_bc, allocstat
1141  ! ..
1142  ! .. Local Arrays ..
1143  integer            descbc( dlen_ )
1144  double precision, dimension ( : ), allocatable :: bc
1145  ! ..
1146  ! .. External Functions ..
1147  integer            indxg2p, numroc
1148  external           indxg2p, numroc
1149  ! ..
1150  ! .. External Subroutines ..
1151  external           blacs_gridinfo, descset, pdgemm, pdlacpy
1152  ! ..
1153  ! .. Executable Statements ..
1154
1155  !
1156  ! Get grid parameters.
1157  !
1158  ictxt = descb( ctxt_ )
1159  call blacs_gridinfo( ictxt, nprow, npcol, myrow, mycol )
1160  nb    = descb( nb_ )
1161
1162  ! Prepare descriptor for copy of matrix b.
1163  proc_row_of_ib_b = indxg2p( ib, descb( mb_ ), myrow, descb( rsrc_ ), nprow )
1164  proc_col_of_jb_b = indxg2p( jb, descb( nb_ ), mycol, descb( csrc_ ), npcol )
1165  mpbc = numroc( m, nb, myrow, proc_row_of_ib_b, nprow )
1166  nqbc = numroc( n, nb, mycol, proc_col_of_jb_b, npcol )
1167  call descset( descbc, m, n, nb, nb, &
1168                proc_row_of_ib_b, proc_col_of_jb_b, ictxt, max( 1, mpbc ) )
1169
1170  ! Allocate copy of matrix b.
1171  len_bc = max( 1, mpbc * nqbc )
1172  allocate( bc( len_bc ), stat = allocstat )
1173  if ( allocstat /= 0 ) stop 'Multiply_BBA: *** Not enough memory for bc ***'
1174
1175  ! Copy b into bc.
1176  call pdlacpy( 'All', m, n, b, ib, jb, descb, &
1177                             bc, 1, 1, descbc )
1178
1179  ! Compute:  b := bc * a.
1180  call pdgemm( transa, transb, m, n, n, &
1181               one, bc, 1, 1, descbc, a, 1, 1, desca, &
1182               zero, b, ib, jb, descb )
1183
1184  ! Deallocate copy of matrix B.
1185  deallocate( bc )
1186
1187  return
1188  !
1189  ! End of Multiply_BBA
1190  !
```

Here is the caller graph for this function:



3.1.1.6 pdgeutv()

```
subroutine pdgeutv (
            integer m,
            integer n,
            double precision, dimension( * ) a,
            integer, dimension( * ) desca,
            logical build_u,
            double precision, dimension( * ) u,
            integer, dimension( * ) descu,
            logical build_v,
            double precision, dimension( * ) v,
            integer, dimension( * ) descv,
            integer q,
            integer info )
```

Computes a randUTV factorization A = U ∗ T ∗ V'.

Parameters

in	*m*	Integer. The number of rows to be operated on, i.e. the number of rows of the distributed matrix A. m >= 0. m >= n.
in	*n*	Integer. The number of columns to be operated on, i.e. the number of columns of the distributed matrix A. n >= 0. m >= n. NOTE: In case m >> n, an initial triangularization of matrix A with the QR factorization and then the randUTV factorization of the triangular factor could improve performances. This technique is used in some linear factorizations such as the SVD when m >> n.
in,out	*a*	Double precision pointer into the local memory to an array, dimension (LLD_a,LOCc(n)) On entry, the local pieces of the m-by-n distributed matrix A which is to be factored. On exit, the elements on and above the diagonal of A contain the min(m,n) by n upper trapezoidal matrix T (T is upper triangular if m >= n); the elements below the diagonal are zeros.
in	*desca*	Integer array, dimension DLEN_ The array descriptor for the distributed matrix A.
in	*build_u*	Logical. If it is true, matrix U must be build.
out	*u*	Double precision pointer into the local memory to an array, dimension (LLD_u,LOCc(m)) On exit, matrix u contains the orthonormal m x m matrix U.
in	*descu*	Integer array, dimension DLEN_ The array descriptor for the distributed matrix U.
in	*build_v*	Logical. If it is true, matrix V must be build.
out	*v*	Double precision pointer into the local memory to an array, dimension (LLD_v,LOCc(n)) On exit, matrix v contains the orthonormal n x n matrix V.
in	*descv*	Integer array, dimension DLEN_ The array descriptor for the distributed matrix V.
in	*q*	Integer. The number of iterations for the iterative power process.
out	*info*	Integer. = 0: Successful exit. < 0: If the i-th argument is an array and the j-entry had an illegal value, then info = -(i∗100+j), if the i-th argument is a scalar and had an illegal value, then info = -i.

Definition at line 61 of file pdgeutv.f90.

```
61   !
62   implicit none
63   !
64   ! .. Scalar Arguments ..
```

```
65   integer            m, n, q, info
66   logical            build_u, build_v
67   ! ..
68   ! .. Array Arguments ..
69   integer            desca( * ), descu( * ), descv( * )
70   double precision   a( * ), u( * ), v( * )
71   ! ..
72   !
73   ! PURPOSE
74   !
75   ! To compute a randUTV factorization
76   !     A = U * T * V'.
77   !
78   ! ARGUMENTS
79   !
80   ! Input/Output Parameters
81   !
82   ! m       (global input) integer
83   !         The number of rows to be operated on, i.e. the number of
84   !         rows of the distributed matrix A. m >= 0. m >= n.
85   !
86   ! n       (global input) integer
87   !         The number of columns to be operated on, i.e. the number
88   !         of columns of the distributed matrix A. n >= 0. m >= n.
89   !         NOTE: In case m » n, an initial triangularization of matrix A
90   !         with the QR factorization and then the randUTV factorization of
91   !         the triangular factor could improve performances. This technique
92   !         is used in some linear factorizations such as the SVD when m » n.
93   !
94   ! a       (local input/local output) double precision pointer into
95   !         the local memory to an array, dimension (LLD_a,LOCc(n))
96   !         On entry, the local pieces of the m-by-n distributed
97   !         matrix A which is to be factored.
98   !         On exit, the elements on and above the diagonal of A
99   !         contain the min(m,n) by n upper trapezoidal matrix T (T
100  !         is upper triangular if m >= n); the elements below the
101  !         diagonal are zeros.
102  !
103  ! desca   (global and local input) integer array, dimension DLEN_
104  !         The array descriptor for the distributed matrix A.
105  !
106  ! build_u (global input) logical
107  !         If it is true, matrix U must be build.
108  !
109  ! u       (local output) double precision pointer into
110  !         the local memory to an array, dimension (LLD_u,LOCc(m))
111  !         On exit, matrix u contains the orthonormal m x m matrix U.
112  !
113  ! descu   (global and local input) integer array, dimension DLEN_
114  !         The array descriptor for the distributed matrix U.
115  !
116  ! build_v (global input) logical
117  !         If it is true, matrix V must be build.
118  !
119  ! v       (local output) double precision pointer into
120  !         the local memory to an array, dimension (LLD_v,LOCc(n))
121  !         On exit, matrix v contains the orthonormal n x n matrix V.
122  !
123  ! descv   (global and local input) integer array, dimension DLEN_
124  !         The array descriptor for the distributed matrix V.
125  !
126  ! q       (global input) integer
127  !         The number of iterations for the iterative power process.
128  !
129  ! info    (global input) integer
130  !
131  ! Error Indicator
132  !
133  ! info    (global output) integer
134  !         = 0: Successful exit.
135  !         < 0: If the i-th argument is an array and the j-entry had
136  !              an illegal value, then info = -(i*100+j), if the
137  !              i-th argument is a scalar and had an illegal value,
138  !              then info = -i.
139  !
140  ! PARALLEL EXECUTION RECOMMENDATIONS
141  !
142  ! Restrictions:
143  ! o The distribution blocks must be square (MB=NB).
144  !
145  ! =====================================================================
146  !
147  ! .. Parameters ..
148  integer            BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DTYPE_, &
149                     LLD_, MB_, M_, NB_, N_, RSRC_
150  parameter( block_cyclic_2d = 1, dlen_ = 9, dtype_ = 1, &
151                       ctxt_ = 2, m_ = 3, n_ = 4, mb_ = 5, nb_ = 6, &
```

```
152                       rsrc_ = 7, csrc_ = 8, lld_ = 9 )
153   double precision    ONE, ZERO
154   parameter( one = 1.0d+0, zero = 0.0d+0 )
155   logical             PROFILE, PRINT_INTERMEDIATE_MATRICES
156   parameter( profile = .false., &
157                       print_intermediate_matrices = .false. )
158   ! ..
159   ! .. Local Scalars ..
160   integer             ictxt, myrow, mycol, nprow, npcol, &
161                       nb, lda, j, k, mn, nbrow, nbcol, &
162                       mpa, nqa, mpg, mpy, nqg, nqy, &
163                       idx_row_of_j, idx_col_of_j, &
164                       proc_row_of_j, proc_col_of_j, &
165                       ii, offjj, allocstat, seed_size
166   double precision    t_normal, t_power, t_rgeqr, t_rormq, t_lgeqr, t_lormq, &
167                       t_svd, t_mma12, t_mma01
168   ! ..
169   ! .. Local Arrays ..
170   integer             descg( DLEN_ ), descy( DLEN_ ), &
171                       descsvdu( DLEN_ ), descsvdvt( DLEN_ ), idum( 1 )
172   double precision, dimension( : ), allocatable :: g, y, vtau, &
173                       svda, svdu, svds, svdvt, s, work_larft
174   integer, dimension( : ), allocatable :: seeds
175   ! ..
176   ! .. External Functions ..
177   integer             numroc
178   external            numroc
179   ! ..
180   ! .. External subroutines ..
181   external            blacs_barrier, blacs_gridinfo, chk1mat, &
182                       compute_local_svd, descset, dlacpy, dlaset, &
183                       generate_normal_matrix, infog2l, Multiply_BAB, &
184                       Multiply_BBA, pchk1mat, pdgemm, pdlarft, pdlaset, &
185                       pdmygeqr2, pdmylarfb_fc, pxerbla, slcombine, sltimer
186   ! ..
187   ! .. Intrinsic Functions ..
188   intrinsic           max, min, random_seed
189   ! ..
190   ! .. Executable Statements ..
191
192   !
193   ! Get grid parameters.
194   !
195   ictxt = desca( ctxt_ )
196   call blacs_gridinfo( ictxt, nprow, npcol, myrow, mycol )
197   lda   = desca( lld_ )
198   nb    = desca( nb_ )
199
200   !
201   ! Check matrix dimensions.
202   !
203   if( m < n ) then
204     if( ( myrow == 0 ).and.( mycol == 0 ) ) then
205       write( *, '(/,1x,a)' ) '*** ERROR in pdgeutv: m should be >= n.'
206       write( *, * ) 'm:  ', m
207       write( *, * ) 'n:  ', n
208     end if
209     info = -1
210     return
211   end if
212
213   !
214   ! Test the input parameters.
215   !
216   info = 0
217   if( nprow == -1 ) then
218     info = -(400+ctxt_)
219   else
220     call chk1mat( m, 1, n, 2, 1, 1, desca, 4, info )
221     if( ( info == 0 ).and. build_u ) then
222       call chk1mat( m, 1, m, 1, 1, 1, descu, 7, info )
223     end if
224     if( ( info == 0 ).and. build_v ) then
225       call chk1mat( n, 2, n, 2, 1, 1, descv, 10, info )
226     end if
227     !
228     ! Check block sizes and alignments.
229     !
230     if( info == 0 ) then
231       if( desca( mb_ ) /= desca( nb_ ) ) then
232         info = -(400+nb_)
233       end if
234       !
235       if( ( info == 0 ) .and. build_u ) then
236         if( ictxt /= descu( ctxt_ ) ) then
237           info = -(700+ctxt_)
238         else if( descu( mb_ ) /= descu( nb_ ) ) then
```

```
239           info = -(700+nb_)
240         else if( desca( mb_ ) /= descu( mb_ ) ) then
241           info = -(700+mb_)
242         else if( desca( rsrc_ ) /= descu( rsrc_ ) ) then
243           info = -(700+rsrc_)
244         else if( desca( csrc_ ) /= descu( csrc_ ) ) then
245           info = -(700+csrc_)
246         end if
247       end if
248       !
249       if( ( info == 0 ) .and. build_v ) then
250         if( ictxt /= descv( ctxt_ ) ) then
251           info = -(1000+ctxt_)
252         else if( descv( mb_ ) /= descv( nb_ ) ) then
253           info = -(1000+nb_)
254         else if( desca( mb_ ) /= descv( mb_ ) ) then
255           info = -(1000+mb_)
256         else if( desca( rsrc_ ) /= descv( rsrc_ ) ) then
257           info = -(1000+rsrc_)
258         else if( desca( csrc_ ) /= descv( csrc_ ) ) then
259           info = -(1000+csrc_)
260         end if
261       end if
262     end if
263     !
264     if( info == 0 ) then
265       call pchk1mat( m, 1, n, 2, 1, 1, desca,  4, 0, idum, idum, info )
266       if ( build_u ) then
267         call pchk1mat( m, 1, m, 1, 1, 1, descu,  7, 0, idum, idum, info )
268       end if
269       if ( build_v ) then
270         call pchk1mat( n, 2, n, 2, 1, 1, descv, 10, 0, idum, idum, info )
271       end if
272     end if
273   end if
274   !
275   if( info /= 0 ) then
276     call pxerbla( ictxt, 'pdgeutv', -info )
277     return
278   end if
279
280   !
281   ! Quick return if possible.
282   !
283   mn = min( m, n )
284   if( mn == 0 ) return
285
286   !
287   ! Allocate local arrays and vectors.
288   !
289   allocate( vtau( n ), stat = allocstat )
290   if ( allocstat /= 0 ) stop 'pdgeutv: *** Not enough memory for vtau ***'
291
292   allocate( svda( nb * nb ), stat = allocstat )
293   if ( allocstat /= 0 ) stop 'pdgeutv: *** Not enough memory for sa ***'
294
295   allocate( svdu( nb * nb ), stat = allocstat )
296   if ( allocstat /= 0 ) stop 'pdgeutv: *** Not enough memory for su ***'
297
298   allocate( svds( nb ), stat = allocstat )
299   if ( allocstat /= 0 ) stop 'pdgeutv: *** Not enough memory for vecs ***'
300
301   allocate( svdvt( nb * nb ), stat = allocstat )
302   if ( allocstat /= 0 ) stop 'pdgeutv: *** Not enough memory for svt ***'
303
304   allocate( s( nb * nb ), stat = allocstat )
305   if ( allocstat /= 0 ) stop 'pdgeutv: *** Not enough memory for s ***'
306
307   allocate( work_larft( nb * nb ), stat = allocstat )
308   if ( allocstat /= 0 ) stop 'pdgeutv: *** Not enough memory for work_larft ***'
309
310   !
311   ! Some initializations.
312   !
313   mpa = numroc( m, nb, myrow, desca( rsrc_ ), nprow )
314   nqa = numroc( n, nb, mycol, desca( csrc_ ), npcol )
315
316   !
317   ! Set different random seeds in every process.
318   !
319   call random_seed( size = seed_size )
320   allocate( seeds( seed_size ) )
321   do k = 1, seed_size
322     !!!! seeds( k ) = k
323     seeds( k ) = 101 + k + myrow + mycol * nprow
324   end do
325   !!!! print *, myrow, mycol, ( seeds( k ), k = 1, seed_size )
```

```
326   call random_seed( put = seeds )
327   deallocate( seeds )
328
329   !
330   ! Set matrices U and V to the identity, if they must be built.
331   !
332   if( build_u ) then
333     call pdlaset( 'All', m, m, zero, one, u, 1, 1, descu )
334   end if
335   if( build_v ) then
336     call pdlaset( 'All', n, n, zero, one, v, 1, 1, descv )
337   end if
338
339   !
340   ! *************************
341   ! * Compute factorization *
342   ! *************************
343   !
344   do j = 1, n, nb
345     nbrow = min( nb, m - j + 1 )
346     nbcol = min( nb, n - j + 1 )
347     call infog2l( j, j, desca, nprow, npcol, myrow, mycol, &
348                   idx_row_of_j, idx_col_of_j, proc_row_of_j, proc_col_of_j )
349
350     if( print_intermediate_matrices ) then
351       if( ( myrow == 0 ).AND.( mycol == 0 ) ) then
352         write ( *, '(/, 40("-"))' )
353         write ( *, * ) 'Iter: ', j, nb, nbrow, nbcol
354         write ( *, '(40("-"), /)' )
355       end if
356     end if
357
358     !
359     ! ================================================================
360     ! Rotate maximal mass of A( :, j:n ) into the current column block.
361     ! ================================================================
362     !
363     ! Perform this processing only if there are more columns to the right of
364     ! the current column block.
365     !
366     if( ( n - j - nbcol + 1 ) > 0 ) then
367       !
368       ! Create matrices G and Y.
369       ! ========================
370       !
371       mpg = numroc( m - j + 1,  nb, myrow, proc_row_of_j, nprow )
372       nqg = numroc( nb, nb, mycol, proc_col_of_j, npcol )
373       mpy = numroc( n - j + 1,  nb, myrow, proc_row_of_j, nprow )
374       nqy = numroc( nb, nb, mycol, proc_col_of_j, npcol )
375
376       call descset( descg, m - j + 1, nb, nb, nb, &
377                     proc_row_of_j, proc_col_of_j, ictxt, max( 1, mpg ) )
378       call descset( descy, n - j + 1, nb, nb, nb, &
379                     proc_row_of_j, proc_col_of_j, ictxt, max( 1, mpy ) )
380
381       allocate( g( max( 1, mpg * nqg ) ), stat = allocstat )
382       if ( allocstat /= 0 ) stop 'pdgeutv: *** Not enough memory for g ***'
383
384       allocate( y( max( 1, mpy * nqy ) ), stat = allocstat )
385       if ( allocstat /= 0 ) stop 'pdgeutv: *** Not enough memory for y ***'
386
387       !
388       ! Generate normal random matrix G.
389       ! ================================
390       !
391       if( profile ) then
392         call blacs_barrier( ictxt, 'All' )
393         call sltimer( 2 )
394       end if
395
396       !!!! call pdlaset( 'All', m - j + 1, nbcol, ZERO, ONE, g, 1, 1, descg )
397       call generate_normal_matrix( m - j + 1, nbcol, g, 1, 1, descg )
398
399       if( profile ) then
400         call sltimer( 2 )
401       end if
402
403       if( print_intermediate_matrices ) then
404         call print_distributed_matrix( m - j + 1, nbcol, g, 1, 1, descg, 'g0' )
405       end if
406
407       !
408       ! Compute the sampling matrix Y.
409       ! ==============================
410       !
411       ! Y = Aloc' * G.
412       !
```

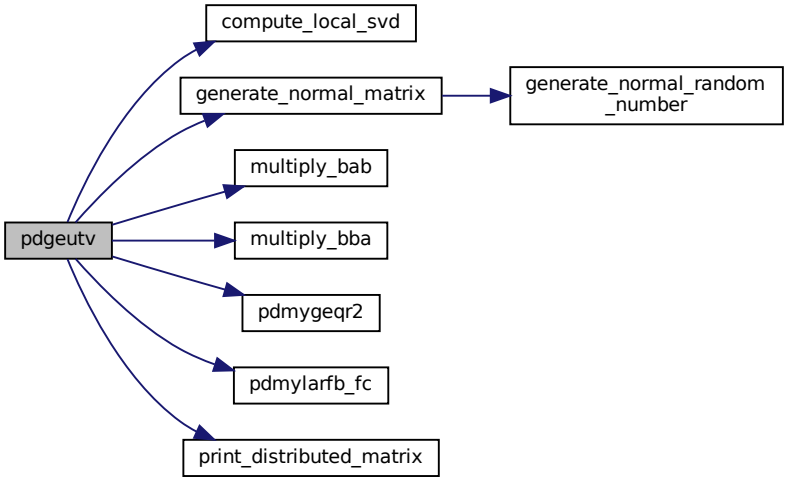
```
413       if( profile ) then
414         call blacs_barrier( ictxt, 'All' )
415         call sltimer( 3 )
416       end if
417
418       call pdgemm( 'Transpose', 'No transpose', n - j + 1, nbcol, m - j + 1, &
419                    one, a, j, j, desca, g, 1, 1, descg, &
420                    zero, y, 1, 1, descy )
421
422       if( print_intermediate_matrices ) then
423         call print_distributed_matrix( n - j + 1, nbcol, y, 1, 1, descy, 'y0' )
424       end if
425
426       !
427       ! Perform the 'power iteration' process.
428       ! ======================================
429       !
430       ! for i_iter = 1:n_iter
431       !   Y = Aloc' * ( Aloc * Y );
432       ! end
433       !
434       do k = 1, q
435         ! Reuse matrix G.
436         call pdgemm( 'No transpose', 'No transpose', &
437                      m - j + 1, nbcol, n - j + 1, &
438                      one, a, j, j, desca, y, 1, 1, descy, &
439                      zero, g, 1, 1, descg )
440         call pdgemm( 'Transpose', 'No transpose', &
441                      n - j + 1, nbcol, m - j + 1, &
442                      one, a, j, j, desca, g, 1, 1, descg, &
443                      zero, y, 1, 1, descy )
444       end do
445
446       if( profile ) then
447         call sltimer( 3 )
448       end if
449
450       if( print_intermediate_matrices ) then
451         call print_distributed_matrix( n - j + 1, nbcol, y, 1, 1, descy, 'y1' )
452       end if
453
454       !
455       ! Update A from the right side. Update V if asked.
456       ! ================================================
457       !
458       ! Construct the Householder transformations to be applied from the right.
459       !
460       if( profile ) then
461         call blacs_barrier( ictxt, 'All' )
462         call sltimer( 4 )
463       end if
464
465       call pdmygeqr2( n - j + 1, nbcol, y, 1, 1, descy, vtau )
466
467       if( profile ) then
468         call sltimer( 4 )
469       end if
470
471       if( print_intermediate_matrices ) then
472         call print_distributed_matrix( n - j + 1, nbcol, y, 1, 1, descy, 'y2' )
473       end if
474
475       !
476       ! Apply the Householder transformations to rotate maximal mass into the
477       ! current column block.
478       ! T(:,[J2,J3])  = T(:,[J2,J3])*Vloc;
479       !
480       if( profile ) then
481         call blacs_barrier( ictxt, 'All' )
482         call sltimer( 5 )
483       end if
484
485       !
486       ! Form the triangular factor of the block reflector.
487       !
488       call pdlarft( 'Forward', 'Columnwise', n - j + 1, nbcol, &
489                     y, 1, 1, descy, vtau, s, work_larft )
490       !
491       ! Apply H to matrix A from the right.
492       !
493       call pdmylarfb_fc( 'Right', 'No transpose', m, n - j + 1, nbcol, &
494                          y, 1, 1, descy, s, a, 1, j, desca )
495
496       !
497       ! Apply H to matrix V from the right.
498       !
499       if( build_v ) then
```

```
500         call pdmylarfb_fc( 'Right', 'No transpose', n, n - j + 1, nbcol, &
501                            y, 1, 1, descy, s, v, 1, j, descv )
502       end if
503
504       ! Deallocate matrices G and Y.
505       deallocate( g )
506       deallocate( y )
507
508       if( profile ) then
509         call sltimer( 5 )
510       end if
511     end if
512
513     if( print_intermediate_matrices ) then
514       call print_distributed_matrix( m, n, a, 1, 1, desca, 'a1' )
515     end if
516
517     !
518     ! =============================================================
519     ! Annihilate elements below the diagonal in current block.
520     ! =============================================================
521     !
522     ! Perform this processing only if there are more rows below the diagonal
523     ! block.
524     !
525     if( ( m - j - nbrow + 1 ) > 0 ) then
526       !
527       ! Update A from the left side. Update U if asked.
528       ! ================================================
529       !
530       ! Determine the rotations to be applied from the left.
531       ! [Uloc,Dloc] = LOCAL_nonpiv_QR(T([J2,I3],J2));
532       !
533       if( profile ) then
534         call blacs_barrier( ictxt, 'All' )
535         call sltimer( 6 )
536       end if
537
538       call pdmygeqr2( m - j + 1, nbcol, a, j, j, desca, vtau )
539
540       if( profile ) then
541         call sltimer( 6 )
542       end if
543
544       if( print_intermediate_matrices ) then
545         call print_distributed_matrix( m, n, a, 1, 1, desca, 'a2' )
546       end if
547
548       !
549       ! Update the rest of matrix A with transformations from the second QR.
550       !
551       if( profile ) then
552         call blacs_barrier( ictxt, 'All' )
553         call sltimer( 7 )
554       end if
555
556       !
557       ! Form the triangular factor of the block reflector.
558       !
559       call pdlarft( 'Forward', 'Columnwise', m - j + 1, nbcol, &
560                     a, j, j, desca, vtau, s, work_larft )
561
562       !
563       ! Apply H' to matrix A from the left.
564       !
565       call pdmylarfb_fc( 'Left', 'Transpose', &
566                          m - j + 1, n - j - nbcol + 1, nbcol, &
567                          a, j, j, desca, s, a, j, j + nbcol, desca )
568
569       !
570       ! Apply H to matrix U from the right.
571       !
572       if( build_u ) then
573         call pdmylarfb_fc( 'Right', 'No transpose', m, m - j + 1, nbcol, &
574                            a, j, j, desca, s, u, 1, j, descu )
575       end if
576
577       if( profile ) then
578         call sltimer( 7 )
579       end if
580
581       if( print_intermediate_matrices ) then
582         call print_distributed_matrix( m, n, a, 1, 1, desca, 'a3' )
583       end if
584
585       !
586       ! Set to zero elements below the diagonal in current block.
```

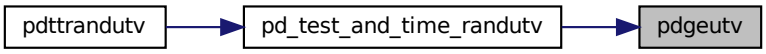
```
587       !
588       call pdlaset( 'Lower', m - j, nbcol, zero, zero, a, j + 1, j, desca )
589
590     end if
591
592     if( print_intermediate_matrices ) then
593       call print_distributed_matrix( m, n, a, 1, 1, desca, 'a4' )
594     end if
595
596     !
597     ! ============================================================
598     ! Compute SVD of diagonal block, and update matrices A, U, and V.
599     ! ============================================================
600     !
601     if( profile ) then
602       call blacs_barrier( ictxt, 'All' )
603       call sltimer( 8 )
604     end if
605
606     !
607     ! The SVD of the diagonal block is computed locally by the process owner
608     ! of the current diagonal block of A.
609     !
610     if( ( myrow == proc_row_of_j ).AND.( mycol == proc_col_of_j ) ) then
611
612       ! Copy current diagonal block of A into local matrix SA.
613       offjj = idx_row_of_j + ( idx_col_of_j - 1 ) * lda
614       call dlacpy( 'All', nbrow, nbcol, a( offjj ), lda, svda, nbrow )
615
616       ! Set current diagonal block to zero.
617       call dlaset( 'All', nbrow, nbcol, zero, zero, a( offjj ), lda )
618
619       ! Compute svd of local matrix SA.
620       !!!! call print_local_matrix( nbrow, nbcol, sa, nbrow, 'sai' )
621       call compute_local_svd( nbrow, nbcol, svda, nbrow, svdu, nbrow, svds, &
622                               svdvt, nbcol, info )
623       !!!! call print_local_matrix( nbrow, nbcol, sa, nbrow, 'saf' )
624       !!!! call print_local_matrix( nbrow, nbrow, su, nbrow, 'suf' )
625       !!!! call print_local_matrix( nbcol, nbcol, svt, nbcol, 'svtf' )
626       !!!! call print_local_matrix( min( nbrow, nbcol ), 1, vecs, nbrow, 'sf' )
627
628       ! Copy back the singular values into the diagonal of the diagonal block.
629       do ii = 1, min( nbrow, nbcol )
630         a( offjj - 1 + ii + ( ii - 1 ) * lda ) = svds( ii )
631       end do
632     end if
633
634     if( profile ) then
635       call sltimer( 8 )
636     end if
637
638     if( profile ) then
639       call blacs_barrier( ictxt, 'All' )
640       call sltimer( 9 )
641     end if
642
643     call descset( descsvdu, nbrow, nbrow, nb, nb, &
644                   proc_row_of_j, proc_col_of_j, ictxt, max( 1, nbrow ) )
645
646     if( print_intermediate_matrices ) then
647       call print_distributed_matrix( m, n, a, 1, 1, desca, 'a5' )
648     end if
649
650     !
651     ! Apply U of miniSVD to A.
652     !
653     if( ( j + nbcol - 1 ) < n ) then
654       call multiply_bab( 'Transpose', 'No transpose', &
655                          nbrow, n - j - nbcol + 1, &
656                          svdu, descsvdu, a, j, j + nbcol, desca )
657     end if
658
659     if( print_intermediate_matrices ) then
660       call print_distributed_matrix( m, n, a, 1, 1, desca, 'a6' )
661     end if
662
663     !
664     ! Apply U of miniSVD to U.
665     !
666     if( build_u ) then
667       call multiply_bba( 'No transpose', 'No transpose', m, nbrow, &
668                          svdu, descsvdu, u, 1, j, descu )
669     end if
670
671     if( profile ) then
672       call sltimer( 9 )
673     end if
```

```
674
675     if( profile ) then
676       call blacs_barrier( ictxt, 'All' )
677       call sltimer( 10 )
678     end if
679
680     call descset( descsvdvt, nbcol, nbcol, nb, nb, &
681                   proc_row_of_j, proc_col_of_j, ictxt, max( 1, nbcol ) )
682
683     if( print_intermediate_matrices ) then
684       call print_distributed_matrix( m, n, a, 1, 1, desca, 'a7' )
685     end if
686
687     !
688     ! Apply V of miniSVD to A.
689     !
690     if( j > 1 ) then
691       call multiply_bba( 'No transpose', 'Transpose', j - 1, nbcol, &
692                          svdvt, descsvdvt, a, 1, j, desca )
693     end if
694
695     if( print_intermediate_matrices ) then
696        call print_distributed_matrix( m, n, a, 1, 1, desca, 'a8' )
697     end if
698
699     !
700     ! Apply V of miniSVD to V.
701     !
702     if( build_v ) then
703       call multiply_bba( 'No transpose', 'Transpose', n, nbcol, &
704                          svdvt, descsvdvt, v, 1, j, descv )
705     end if
706
707     if( profile ) then
708       call sltimer( 10 )
709     end if
710
711     if( print_intermediate_matrices ) then
712       call print_distributed_matrix( m, n, a, 1, 1, desca, 'a9' )
713     end if
714   end do
715
716   ! Deallocate local arrays and vectors.
717   deallocate( vtau )
718   deallocate( svda )
719   deallocate( svdu )
720   deallocate( svds )
721   deallocate( svdvt )
722   deallocate( s )
723   deallocate( work_larft )
724
725   ! Compute and print final timings, if needed.
726   if( profile ) then
727     ! Gather maximum of all CPU and WALL clock timings.
728     call slcombine( ictxt, 'All', '>', 'w', 1, 2, t_normal )
729     call slcombine( ictxt, 'All', '>', 'w', 1, 3, t_power )
730     call slcombine( ictxt, 'All', '>', 'w', 1, 4, t_rgeqr )
731     call slcombine( ictxt, 'All', '>', 'w', 1, 5, t_rormq )
732     call slcombine( ictxt, 'All', '>', 'w', 1, 6, t_lgeqr )
733     call slcombine( ictxt, 'All', '>', 'w', 1, 7, t_lormq )
734     call slcombine( ictxt, 'All', '>', 'w', 1, 8, t_svd )
735     call slcombine( ictxt, 'All', '>', 'w', 1, 9, t_mma12 )
736     call slcombine( ictxt, 'All', '>', 'w', 1, 10, t_mma01 )
737
738     if( ( myrow == 0 ).and.( mycol == 0 ) ) then
739       write ( *, '(/,1x,a)' ) 'Profiling of pdgeutv (time in s.): '
740       write ( *, '(1x,a,f12.4)' ) '  t_normal: ', t_normal
741       write ( *, '(1x,a,f12.4)' ) '  t_power:  ', t_power
742       write ( *, '(1x,a,f12.4)' ) '  t_rgeqr:  ', t_rgeqr
743       write ( *, '(1x,a,f12.4)' ) '  t_rormq:  ', t_rormq
744       write ( *, '(1x,a,f12.4)' ) '  t_lgeqr:  ', t_lgeqr
745       write ( *, '(1x,a,f12.4)' ) '  t_lormq:  ', t_lormq
746       write ( *, '(1x,a,f12.4)' ) '  t_svd:    ', t_svd
747       write ( *, '(1x,a,f12.4)' ) '  t_mma12:  ', t_mma12
748       write ( *, '(1x,a,f12.4)' ) '  t_mma01:  ', t_mma01
749       write ( *, '(1x,a,f12.4)' ) '  t_total:  ', &
750           t_normal + t_power + t_rgeqr + t_rormq + t_lgeqr + t_lormq + &
751           t_svd + t_mma12 + t_mma01
752       write ( *, '(1x,a)' ) 'End of profiling'
753     end if
754   end if
755
756   return
757   ! *** Last line of pdgeutv ***
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.1.1.7 pdmygeqr2()

```
subroutine pdmygeqr2 (
             integer m,
             integer n,
             double precision, dimension( * ) a,
             integer ia,
             integer ja,
             integer, dimension( * ) desca,
             double precision, dimension( * ) tau )
```

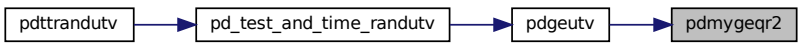
Computes the QR factorization of a real distributed m-by-n matrix
sub(A) = A(IA:IA+M-1,JA:JA+N-1) = Q ∗ R.

Definition at line 764 of file pdgeutv.f90.

```
764   !
765   implicit none
766   !
767   ! .. Scalar Arguments ..
768   integer            ia, ja, m, n
```

```
769   ! ..
770   ! .. Array Arguments ..
771   integer            desca( * )
772   double precision   a( * ), tau( * )
773   ! ..
774   !
775   !  Purpose
776   !  =======
777   !
778   !  It computes the QR factorization of a real distributed m-by-n matrix
779   !  sub( A ) = A(IA:IA+M-1,JA:JA+N-1) = Q * R.
780   !
781   !  ======================================================================
782   !
783   ! .. Local Scalars ..
784   double precision   scalar_work
785   integer            info, len_work, allocstat
786   ! ..
787   ! .. Local Arrays ..
788   double precision, dimension ( : ), allocatable :: work
789   ! ..
790   ! .. External Subroutines ..
791   external           pdgeqr2
792   ! ..
793   ! .. Intrinsic Functions ..
794   intrinsic          int
795   ! ..
796   ! .. Executable Statements ..
797
798   !
799   ! Compute workspace length.
800   !
801   call pdgeqr2( m, n, a, ia, ja, desca, tau, scalar_work, -1, info )
802   if ( info /= 0 ) then
803     write ( *, * ) &
804         '*** ERROR in pdmygeqr2: Info of call to compute wk length: ', info
805     stop
806   end if
807   len_work = int( scalar_work )
808
809   !
810   ! Allocate workspace.
811   !
812   allocate( work( len_work ), stat = allocstat )
813   if ( allocstat /= 0 ) stop 'pdmygeqr2: *** Not enough memory for work ***'
814
815   !
816   ! Call to pdgeqr2.
817   !
818   call pdgeqr2( m, n, a, ia, ja, desca, tau, work, len_work, info )
819   if ( info /= 0 ) then
820     write ( *, * ) '*** ERROR in pdmygeqr2: Info of pdgeqr2: ', info
821     stop
822   end if
823
824   !
825   ! Deallocate workspace.
826   !
827   deallocate( work )
828
829   return
830   !
831   ! End of pdmygeqr2
832   !
```

Here is the caller graph for this function:



3.1.1.8 pdmylarfb_fc()

```
subroutine pdmylarfb_fc (
             character side,
             character trans,
             integer m,
             integer n,
             integer k,
             double precision, dimension( * ) v,
             integer iv,
             integer jv,
             integer, dimension( * ) descv,
             double precision, dimension( * ) t,
             double precision, dimension( * ) c,
             integer ic,
             integer jc,
             integer, dimension( * ) descc )
```

Applies a real block reflector Q or its transpose Q**T to a real distributed M-by-N matrix sub(C) = C(IC:IC+↩ M-1,JC:JC+N-1) from the left or from the right. Matrix Q is intrisically stored in matrix v (Householder vectors) and vector t (tau factors).

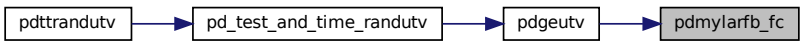
Definition at line 844 of file pdgeutv.f90.

```
844   !
845   implicit none
846   !
847   ! .. Scalar Arguments ..
848   character          side, trans
849   integer            ic, iv, jc, jv, k, m, n
850   ! ..
851   ! .. Array Arguments ..
852   integer            descc( * ), descv( * )
853   double precision   c( * ), t( * ), v( * )
854   ! ..
855   !
856   !  Purpose
857   !  =======
858   !
859   !  It applies a real block reflector Q or its transpose Q**T to a real
860   !  distributed M-by-N matrix sub( C ) = C(IC:IC+M-1,JC:JC+N-1) from the
861   !  left or from the right.
862   !  Matrix Q is intrisically stored in matrix v (Householder vectors) and
863   !  vector t (tau factors).
864   !
865   !  Workspace required inside this function
866   !  =======================================
867   !
868   !  WORK    (local workspace) double precision array, dimension (LWORK)
869   !          If STOREV = 'C',
870   !            if SIDE = 'L',
871   !              LWORK >= ( NqC0 + MpC0 ) * K
872   !            else if SIDE = 'R',
873   !              LWORK >= ( NqC0 + MAX( NpV0 + NUMROC( NUMROC( N+ICOFFC,
874   !                          NB_V, 0, 0, NPCOL ), NB_V, 0, 0, LCMQ ),
875   !                          MpC0 ) ) * K
876   !            end if
877   !          else if STOREV = 'R',
878   !            if SIDE = 'L',
879   !              LWORK >= ( MpC0 + MAX( MqV0 + NUMROC( NUMROC( M+IROFFC,
880   !                          MB_V, 0, 0, NPROW ), MB_V, 0, 0, LCMP ),
881   !                          NqC0 ) ) * K
882   !            else if SIDE = 'R',
883   !              LWORK >= ( MpC0 + NqC0 ) * K
884   !            end if
885   !          end if
886   !
887   !          where LCMQ = LCM / NPCOL with LCM = ICLM( NPROW, NPCOL ),
888   !
889   !          IROFFV = MOD( IV-1, MB_V ), ICOFFV = MOD( JV-1, NB_V ),
890   !          IVROW = INDXG2P( IV, MB_V, MYROW, RSRC_V, NPROW ),
891   !          IVCOL = INDXG2P( JV, NB_V, MYCOL, CSRC_V, NPCOL ),
892   !          MqV0 = NUMROC( M+ICOFFV, NB_V, MYCOL, IVCOL, NPCOL ),
893   !          NpV0 = NUMROC( N+IROFFV, MB_V, MYROW, IVROW, NPROW ),
894   !
895   !          IROFFC = MOD( IC-1, MB_C ), ICOFFC = MOD( JC-1, NB_C ),
```

```
896   !          ICROW = INDXG2P( IC, MB_C, MYROW, RSRC_C, NPROW ),
897   !          ICCOL = INDXG2P( JC, NB_C, MYCOL, CSRC_C, NPCOL ),
898   !          MpC0 = NUMROC( M+IROFFC, MB_C, MYROW, ICROW, NPROW ),
899   !          NpC0 = NUMROC( N+ICOFFC, MB_C, MYROW, ICROW, NPROW ),
900   !          NqC0 = NUMROC( N+ICOFFC, NB_C, MYCOL, ICCOL, NPCOL ),
901   !
902   !          ILCM, INDXG2P and NUMROC are ScaLAPACK tool functions;
903   !          MYROW, MYCOL, NPROW and NPCOL can be determined by calling
904   !          the subroutine BLACS_GRIDINFO.
905   !
906   !  =====================================================================
907   !
908   ! .. Parameters ..
909   integer            BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DTYPE_, &
910                      LLD_, MB_, M_, NB_, N_, RSRC_
911   parameter( block_cyclic_2d = 1, dlen_ = 9, dtype_ = 1, &
912                        ctxt_ = 2, m_ = 3, n_ = 4, mb_ = 5, nb_ = 6, &
913                        rsrc_ = 7, csrc_ = 8, lld_ = 9 )
914   double precision   ONE, ZERO
915   parameter( one = 1.0d+0, zero = 0.0d+0 )
916   ! ..
917   ! .. Local Scalars ..
918   double precision   scalar_work
919   integer            ictxt, myrow, mycol, nprow, npcol, &
920                      INFO, LWORK, LCM, LCMQ, &
921                      IROFFV, ICOFFV, IVROW, IVCOL, MqV0, NpV0, &
922                      IROFFC, ICOFFC, ICROW, ICCOL, MpC0, NpC0, NqC0, &
923                      RSRC_C, CSRC_C, RSRC_V, CSRC_V, &
924                      MB_C, NB_C, MB_V, NB_V, &
925                      len_work, allocstat
926   ! ..
927   ! .. Local Arrays ..
928   double precision, dimension ( : ), allocatable :: work
929   ! ..
930   ! .. External Functions ..
931   logical            lsame
932   integer            ilcm, indxg2p, numroc
933   external           ilcm, indxg2p, numroc, lsame
934   ! ..
935   ! .. External subroutines ..
936   external           blacs_gridinfo, pdlarfb, pdormqr
937   ! ..
938   ! .. Intrinsic Functions ..
939   intrinsic          max, min, mod
940   ! ..
941   ! .. Executable Statements ..
942
943   !
944   ! Get grid parameters.
945   !
946   ictxt = descc( ctxt_ )
947   call blacs_gridinfo( ictxt, nprow, npcol, myrow, mycol )
948
949   !
950   ! Compute workspace for pdlarfb with arguments 'Forward' and 'Columnwise'.
951   !
952   rsrc_c = descc( rsrc_ )
953   csrc_c = descc( csrc_ )
954   rsrc_v = descv( rsrc_ )
955   csrc_v = descv( csrc_ )
956   mb_c   = descc( mb_ )
957   nb_c   = descc( nb_ )
958   mb_v   = descv( mb_ )
959   nb_v   = descv( nb_ )
960   !
961   lcm = ilcm( nprow, npcol )
962   lcmq = lcm / npcol
963   !
964   iroffv = mod( iv-1, mb_v )
965   icoffv = mod( jv-1, nb_v )
966   ivrow = indxg2p( iv, mb_v, myrow, rsrc_v, nprow )
967   ivcol = indxg2p( jv, nb_v, mycol, csrc_v, npcol )
968   mqv0 = numroc( m+icoffv, nb_v, mycol, ivcol, npcol )
969   npv0 = numroc( n+iroffv, mb_v, myrow, ivrow, nprow )
970   !
971   iroffc = mod( ic-1, mb_c )
972   icoffc = mod( jc-1, nb_c )
973   icrow = indxg2p( ic, mb_c, myrow, rsrc_c, nprow )
974   iccol = indxg2p( jc, nb_c, mycol, csrc_c, npcol )
975   mpc0 = numroc( m+iroffc, mb_c, myrow, icrow, nprow )
976   npc0 = numroc( n+icoffc, mb_c, myrow, icrow, nprow )
977   nqc0 = numroc( n+icoffc, nb_c, mycol, iccol, npcol )
978   !
979   if( lsame( side, 'L' ) ) then
980     lwork = ( nqc0 + mpc0 ) * k
981   elseif( lsame( side, 'R' ) ) then
982     lwork = ( nqc0 + max( npv0 + numroc( numroc( n+icoffc, &
```

```
983                nb_v, 0, 0, npcol ), nb_v, 0, 0, lcmq ), &
984                mpc0 ) ) * k
985    else
986      lwork = 0
987    end if
988
989    !
990    ! Compute workspace required by pdormqr, and compare with the one of pdlarfb.
991    ! The workspace length of pdormqr should be larger than that of pdlarfb.
992    !
993    call pdormqr( side, trans, m, n, k, &
994                  v, iv, jv, descv, t, c, ic, jc, descc, &
995                  scalar_work, -1, info )
996    len_work = int( scalar_work )
997    if ( len_work < lwork ) stop '*** ERROR in pdmylarfb_fc: len_work < LWORK.'
998
999    !
1000    ! Allocate workspace.
1001    !
1002    len_work = lwork
1003    allocate( work( len_work ), stat = allocstat )
1004    if ( allocstat /= 0 ) stop 'pdmylarfb_fc: *** Not enough memory for work ***'
1005
1006    !
1007    ! Call to pdlarfb.
1008    !
1009    call pdlarfb( side, trans, 'Forward', 'Columnwise', m, n, k, v, iv, &
1010                  jv, descv, t, c, ic, jc, descc, work )
1011
1012    !
1013    ! Deallocate workspace.
1014    !
1015    deallocate( work )
1016
1017    return
1018    !
1019    ! End of pdmylarfb_fc
1020    !
```

Here is the caller graph for this function:



3.2 pdttrandutv.f90 File Reference

Functions/Subroutines

- program pdttrandutv

  Main driver for randUTV computation.
- subroutine pd_test_and_time_randutv (ictxt, m, n, nb, q_factor, iseed, print_matrices, wtime, resid)

  Tests and times the randUTV factorization.
- double precision function check_utv_resids (m, n, ac, descac, u, descu, t, desct, v, descv)

  It checks the residuals of the randUTV factorization.
- double precision function compar_svd (m, n, a1, desca1, a2, desca2)

  Computes: || *singular_values(a1) - singular_values(a2)* || / || *singular_values(a1)* ||.
- subroutine generate_random_matrix (matrix_type, m, n, iseed, a, desca)

  It generates random double-precision m-by-n matrix a.
- subroutine print_distributed_matrix (m, n, a, ia, ja, desca, cname)

  Prints the distributed matrix.
- subroutine print_local_matrix (m, n, a, lda, cname)

  Prints the local matrix in Matlab/Octave format.

3.2.1 Function/Subroutine Documentation

3.2.1.1 check_utv_resids()

```
double precision function check_utv_resids (
            integer m,
            integer n,
            double precision, dimension( * ) ac,
            integer, dimension( * ) descac,
            double precision, dimension( * ) u,
            integer, dimension( * ) descu,
            double precision, dimension( * ) t,
            integer, dimension( * ) desct,
            double precision, dimension( * ) v,
            integer, dimension( * ) descv )
```

It checks the residuals of the randUTV factorization.

Definition at line 345 of file pdttrandutv.f90.

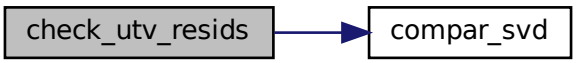
```
345   !
346   implicit none
347   !
348   ! .. Scalar arguments ..
349   integer            m, n
350   ! ..
351   ! .. Array arguments ..
352   integer            descac( * ), descu( * ), desct( * ), descv( * )
353   double precision   ac( * ), u( * ), t( * ), v( * )
354   ! ..
355   !
356   ! Purpose
357   ! =======
358   !
359   ! It checks the residuals of the randUTV factorization.
360   !
361   ! =====================================================================
362   !
363   ! .. Parameters ..
364   integer            BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DT_, &
365                      LLD_, MB_, M_, NB_, N_, RSRC_
366   parameter( block_cyclic_2d = 1, dlen_ = 9, dt_ = 1, &
367                        ctxt_ = 2, m_ = 3, n_ = 4, mb_ = 5, nb_ = 6, &
368                        rsrc_ = 7, csrc_ = 8, lld_ = 9 )
369   double precision   ONE, ZERO
370   parameter( one = 1.0d+0, zero = 0.0d+0 )
371   ! ..
372   ! .. Local arrays ..
373   integer            descut( DLEN_ ), desctt( DLEN_ ), &
374                      descmm( DLEN_ ), descmn( DLEN_ ), descnn( DLEN_ )
375   double precision, dimension ( : ), allocatable :: ut, tt, mm, mn, nn
376   ! ..
377   ! .. Local scalars ..
378   double precision   nrm, resida, residb, residc, residd, reside, residf, &
379                      maxres
380   integer            ictxt, nb, nprow, npcol, myrow, mycol, &
381                      info, allocstat, &
382                      nput, nqut, nptt, nqtt, &
383                      npmm, nqmm, npmn, nqmn, npnn, nqnn, dummy
384   ! ..
385   ! .. External Functions ..
386   integer            numroc
387   double precision   pdlange, compar_svd
388   external           numroc, pdlange, compar_svd
389   ! ..
390   ! .. External Subroutines ..
391   external           blacs_gridinfo, descinit, pdgemm, pdlacpy, pdlaset
392   ! ..
393   ! .. Intrinsic Functions ..
394   intrinsic          int, max
395   ! ..
396   ! .. Executable Statements ..
397
```

```
398   !
399   ! Get grid parameters.
400   !
401   ictxt = descac( ctxt_ )
402   call blacs_gridinfo( ictxt, nprow, npcol, myrow, mycol )
403   nb    = descac( nb_ )
404
405   !
406   ! Some initializations.
407   !
408   nput  = numroc( m, nb, myrow, 0, nprow )
409   nqut  = numroc( n, nb, mycol, 0, npcol )
410   nptt  = numroc( m, nb, myrow, 0, nprow )
411   nqtt  = numroc( n, nb, mycol, 0, npcol )
412   npmm  = numroc( m, nb, myrow, 0, nprow )
413   nqmm  = numroc( m, nb, mycol, 0, npcol )
414   npmn  = numroc( m, nb, myrow, 0, nprow )
415   nqmn  = numroc( n, nb, mycol, 0, npcol )
416   npnn  = numroc( n, nb, myrow, 0, nprow )
417   nqnn  = numroc( n, nb, mycol, 0, npcol )
418
419   !
420   ! Initialize array descriptors for the matrices MM, and MN.
421   !
422   call descinit( descut, m, n, nb, nb, 0, 0, ictxt, max( 1, nput ), info )
423   call descinit( desctt, m, n, nb, nb, 0, 0, ictxt, max( 1, nptt ), info )
424   call descinit( descmm, m, m, nb, nb, 0, 0, ictxt, max( 1, npmm ), info )
425   call descinit( descmn, m, n, nb, nb, 0, 0, ictxt, max( 1, npmn ), info )
426   call descinit( descnn, n, n, nb, nb, 0, 0, ictxt, max( 1, npnn ), info )
427
428   !
429   ! Allocate local arrays.
430   !
431   allocate( ut( nput * nqut ), stat = allocstat )
432   if ( allocstat /= 0 ) stop '*** Not enough memory for ut ***'
433
434   allocate( tt( nptt * nqtt ), stat = allocstat )
435   if ( allocstat /= 0 ) stop '*** Not enough memory for tt ***'
436
437   allocate( mm( npmm * nqmm ), stat = allocstat )
438   if ( allocstat /= 0 ) stop '*** Not enough memory for mm ***'
439
440   allocate( mn( npmn * nqmn ), stat = allocstat )
441   if ( allocstat /= 0 ) stop '*** Not enough memory for mn ***'
442
443   allocate( nn( npnn * nqnn ), stat = allocstat )
444   if ( allocstat /= 0 ) stop '*** Not enough memory for nn ***'
445
446   !
447   ! Print initial message.
448   !
449   if( ( myrow == 0 ).and.( mycol == 0 ) ) then
450     write( *, * )
451     write( *, '(/,1x,a)' ) 'Residuals of randUTV factorization'
452   end if
453
454   !
455   ! Compute residual: || Ac - U * triu( T ) * V' || / || Ac ||.
456   !
457   call pdlacpy( 'All', m, n, t, 1, 1, desct, &
458                              tt, 1, 1, desctt )
459   call pdlaset( 'Lower', m - 1, n, zero, zero, tt, 2, 1, desctt )
460
461   call pdgemm( 'No transpose', 'No transpose', m, n, m, &
462                one, u, 1, 1, descu, tt, 1, 1, desctt, &
463                zero,  ut, 1, 1, descut )
464
465   call pdlacpy( 'All', m, n, ac, 1, 1, descac, &
466                              mn, 1, 1, descmn )
467   call pdgemm( 'No transpose', 'Transpose', m, n, n, &
468                -one, ut, 1, 1, descut, v, 1, 1, descv, &
469                one,  mn, 1, 1, descmn )
470   !
471   resida = pdlange( 'Frobenius', m, n, mn, 1, 1, descmn, dummy )
472   nrm = pdlange( 'Frobenius', m, n, ac, 1, 1, descac, dummy )
473   if( nrm /= zero ) then
474     resida = resida / nrm
475   end if
476
477   if( ( myrow == 0 ).and.( mycol == 0 ) ) then
478     write( *, '(1x,a,e12.4)' ) &
479         '|| Ac - U * triu( T ) * Vt || / || Ac || =  ', resida
480   end if
481
482   !
483   ! Compute residual: || I - U' * U || / || U ||.
484   !
```

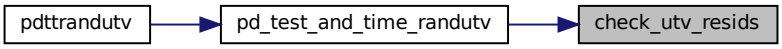
```
485   call pdlaset( 'All', m, m, zero, one, mm, 1, 1, descmm )
486   call pdgemm( 'Transpose', 'No transpose', m, m, m, &
487                -one, u, 1, 1, descu, u, 1, 1, descu, &
488                one,  mm, 1, 1, descmm )
489   !
490   residb = pdlange( 'Frobenius', m, m, mm, 1, 1, descmm, dummy )
491   nrm = pdlange( 'Frobenius', m, m, u, 1, 1, descu, dummy )
492   if( nrm /= zero ) then
493     residb = residb / nrm
494   end if
495
496   if( ( myrow == 0 ).and.( mycol == 0 ) ) then
497     write( *, '(1x,a,e12.4)' ) &
498         '|| I - Ut * U || / || U || =                ', residb
499   end if
500
501   !
502   ! Compute residual: || I - U * U' || / || U ||.
503   !
504   call pdlaset( 'All', m, m, zero, one, mm, 1, 1, descmm )
505   call pdgemm( 'No transpose', 'Transpose', m, m, m, &
506                -one, u, 1, 1, descu, u, 1, 1, descu, &
507                one,  mm, 1, 1, descmm )
508   !
509   residc = pdlange( 'Frobenius', m, m, mm, 1, 1, descmm, dummy )
510   nrm = pdlange( 'Frobenius', m, m, u, 1, 1, descu, dummy )
511   if( nrm /= zero ) then
512     residc = residc / nrm
513   end if
514
515   if( ( myrow == 0 ).and.( mycol == 0 ) ) then
516     write( *, '(1x,a,e12.4)' ) &
517         '|| I - U * Ut || / || U || =                ', residc
518   end if
519
520   !
521   ! Compute residual: || I - V' * V || / || V ||.
522   !
523   call pdlaset( 'All', n, n, zero, one, nn, 1, 1, descnn )
524   call pdgemm( 'Transpose', 'No transpose', n, n, n, &
525                -one, v, 1, 1, descv, v, 1, 1, descv, &
526                one,  nn, 1, 1, descnn )
527   !
528   residd = pdlange( 'Frobenius', n, n, nn, 1, 1, descnn, dummy )
529   nrm = pdlange( 'Frobenius', n, n, v, 1, 1, descv, dummy )
530   if( nrm /= zero ) then
531     residd = residd / nrm
532   end if
533
534   if( ( myrow == 0 ).and.( mycol == 0 ) ) then
535     write( *, '(1x,a,e12.4)' ) &
536         '|| I - Vt * V || / || V || =                ', residd
537   end if
538
539   !
540   ! Compute residual: || I - V * V' || / || V ||.
541   !
542   call pdlaset( 'All', n, n, zero, one, nn, 1, 1, descnn )
543   call pdgemm( 'No transpose', 'Transpose', n, n, n, &
544                -one, v, 1, 1, descv, v, 1, 1, descv, &
545                one,  nn, 1, 1, descnn )
546   !
547   reside = pdlange( 'Frobenius', n, n, nn, 1, 1, descnn, dummy )
548   nrm = pdlange( 'Frobenius', n, n, v, 1, 1, descv, dummy )
549   if( nrm /= zero ) then
550     reside = reside / nrm
551   end if
552
553   if( ( myrow == 0 ).and.( mycol == 0 ) ) then
554     write( *, '(1x,a,e12.4)' ) &
555         '|| I - V * Vt || / || V || =                ', reside
556   end if
557
558   !
559   ! Compute residual: || singular_values( ac ) - singular_values( triu( t ) ||
560   !                   / || singular_values( ac ) ||.
561   !
562   residf = compar_svd( m, n, ac, descac, t, desct )
563
564   if( ( myrow == 0 ).and.( mycol == 0 ) ) then
565     write( *, '(1x,a,e12.4)' ) &
566         '|| svd( a ) - svd( t )|| / || svd( a ) || = ', residf
567   end if
568
569   !
570   ! Deallocate local arrays and vectors.
571   !
```

```
572   deallocate( ut )
573   deallocate( tt )
574   deallocate( mm )
575   deallocate( mn )
576   deallocate( nn )
577
578   !
579   ! Compute and print maximum residual.
580   !
581   maxres = max( resida, residb, residc, residd, reside, residf )
582
583   if( ( myrow == 0 ).and.( mycol == 0 ) ) then
584     write( *, * )
585     write( *, '(1x,a,e12.4)' ) 'Maximum residual: ', maxres
586   end if
587
588   check_utv_resids = maxres
589   return
590   !
591   ! End of check_utv_resids
592   !
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.2.1.2 compar_svd()

```
double precision function compar_svd (
             integer m,
             integer n,
             double precision, dimension( * ) a1,
             integer, dimension( * ) desca1,
             double precision, dimension( * ) a2,
             integer, dimension( * ) desca2 )
```

Computes: || singular_values(a1) - singular_values(a2) || / || singular_values(a1) ||.

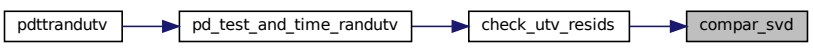
Definition at line 600 of file pdttrandutv.f90.

```
600   !
601   implicit none
602   !
```

```
603   ! .. Scalar Arguments ..
604   integer            m, n
605   ! ..
606   ! .. Array Arguments ..
607   integer            desca1( * ), desca2( * )
608   double precision   a1( * ), a2( * )
609   ! ..
610   !
611   ! Purpose
612   ! =======
613   !
614   ! It computes the following:
615   !   || singular_values( a1 ) - singular_values( a2 ) || /
616   !   || singular_values( a1 ) ||.
617   !
618   ! ======================================================================
619   !
620   ! .. Parameters ..
621   integer            BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DTYPE_, &
622                      LLD_, MB_, M_, NB_, N_, RSRC_
623   parameter( block_cyclic_2d = 1, dlen_ = 9, dtype_ = 1, &
624                        ctxt_ = 2, m_ = 3, n_ = 4, mb_ = 5, nb_ = 6, &
625                        rsrc_ = 7, csrc_ = 8, lld_ = 9 )
626   double precision   ONE, ZERO
627   parameter( one = 1.0d+0, zero = 0.0d+0 )
628   ! ..
629   ! .. Local Arrays ..
630   double precision, dimension ( : ), allocatable :: ac, vs1, vs2, work
631   integer            descac( dlen_ )
632   ! ..
633   ! .. Local Scalars ..
634   integer            ictxt, nprow, npcol, myrow, mycol, &
635                      nb, npac, nqac, allocstat, len_work, info
636   double precision   scalar_work, nrm, res
637   ! ..
638   ! .. External Functions ..
639   integer            numroc
640   double precision   dnrm2
641   external           numroc, dnrm2
642   ! ..
643   ! .. External Subroutines ..
644   external           blacs_gridinfo, daxpy, descinit, pdgesvd, pdlacpy
645   ! ..
646   ! .. Intrinsic Functions ..
647   intrinsic          int, max
648   ! ..
649   ! .. Executable Statements ..
650
651   !
652   ! Get grid parameters.
653   !
654   ictxt = desca1( ctxt_ )
655   call blacs_gridinfo( ictxt, nprow, npcol, myrow, mycol )
656   nb    = desca1( nb_ )
657
658   !
659   ! Some initializations.
660   !
661   npac  = numroc( m, nb, myrow, 0, nprow )
662   nqac  = numroc( n, nb, mycol, 0, npcol )
663   call descinit( descac, m, n, nb, nb, 0, 0, ictxt, max( 1, npac ), info )
664
665   !
666   ! Compute optimal real workspace length.
667   !
668   call pdgesvd( 'None', 'None', m, n, a1, 1, 1, desca1, &
669                 vs1, a1, 1, 1, desca1, a1, 1, 1, desca1, &
670                 scalar_work, -1, info )
671   if( info /= 0 ) then
672     if( ( myrow == 0 ).and.( mycol == 0 ) ) then
673       write ( *, * ) '*** Error in compar_svd:', &
674                      'Info after query-request  pdgesvd: ', info
675     end if
676   end if
677   len_work = int( scalar_work )
678
679   !
680   ! Allocate local arrays and vectors.
681   !
682   allocate( ac( npac * nqac ), stat = allocstat )
683   if ( allocstat /= 0 ) stop '*** Not enough memory for ac ***'
684
685   allocate( vs1( min( m, n ) ), stat = allocstat )
686   if ( allocstat /= 0 ) stop '*** Not enough memory for vs1 ***'
687
688   allocate( vs2( min( m, n ) ), stat = allocstat )
689   if ( allocstat /= 0 ) stop '*** Not enough memory for vs2 ***'
```

```
690
691   allocate( work( len_work ), stat = allocstat )
692   if ( allocstat /= 0 ) stop '*** Not enough memory for work ***'
693
694   !
695   ! Compute singular values of a1.
696   !
697   call pdlacpy( 'All', m, n, a1, 1, 1, desca1, &
698                              ac, 1, 1, descac )
699   call pdgesvd( 'None', 'None', m, n, ac, 1, 1, descac, &
700                 vs1, ac, 1, 1, descac, ac, 1, 1, descac, &
701                 work, len_work, info )
702   if( info /= 0 ) THEN
703     if( ( myrow == 0 ).and.( mycol == 0 ) ) then
704       write ( *, * ) '*** Error in compar_svd:', &
705                      'Info after first pdgesvd: ', info
706     end if
707   end if
708
709   !
710   ! Compute singular values of a2.
711   !
712   call pdlacpy( 'All', m, n, a2, 1, 1, desca2, &
713                              ac, 1, 1, descac )
714   call pdgesvd( 'None', 'None', m, n, ac, 1, 1, descac, &
715                 vs2, ac, 1, 1, descac, ac, 1, 1, descac, &
716                 work, len_work, info )
717   if( info /= 0 ) then
718     if( ( myrow == 0 ).and.( mycol == 0 ) ) then
719       write ( *, * ) '*** Error in compar_svd:', &
720                      'Info after second pdgesvd: ', info
721     end if
722   end if
723
724   !
725   ! Compute the norm of the singular values.
726   !
727   nrm = dnrm2( min( m, n ), vs1, 1 )
728
729   !
730   ! Compute the norm of the difference between the singular values.
731   !
732   call daxpy( min( m, n ), - one, vs2, 1, vs1, 1 )
733   res = dnrm2( min( m, n ), vs1, 1 )
734
735   !
736   ! Compute the relative residual.
737   !
738   if( nrm /= zero ) then
739     compar_svd = res / nrm
740   else
741     compar_svd = res
742   end if
743   !
744   ! Deallocate local arrays and vectors.
745   !
746   deallocate( ac )
747   deallocate( vs1 )
748   deallocate( vs2 )
749   deallocate( work )
750
751   return
752   !
753   ! End of compar_svd
754   !
```

Here is the caller graph for this function:



3.2.1.3 generate_random_matrix()

```
subroutine generate_random_matrix (
             integer matrix_type,
             integer m,
             integer n,
             integer, dimension( * ) iseed,
             double precision, dimension( * ) a,
             integer, dimension( * ) desca )
```

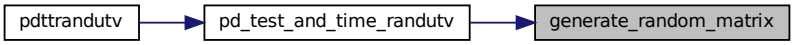
It generates random double-precision m-by-n matrix a.

Definition at line 760 of file pdttrandutv.f90.

```
760   !
761   implicit none
762   !
763   ! .. Scalar Arguments ..
764   integer            matrix_type, m, n
765   ! ..
766   ! .. Array Arguments ..
767   integer            desca( * ), iseed( * )
768   double precision   a( * )
769   ! ..
770   !
771   ! Purpose
772   ! =======
773   !
774   ! It generates random double-precision m-by-n matrix a.
775   ! This code should not be used for large matrices since it is not
776   ! efficient.
777   !
778   ! =====================================================================
779   !
780   ! .. Parameters ..
781   integer            BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DTYPE_, &
782                      LLD_, MB_, M_, NB_, N_, RSRC_
783   parameter( block_cyclic_2d = 1, dlen_ = 9, dtype_ = 1, &
784                        ctxt_ = 2, m_ = 3, n_ = 4, mb_ = 5, nb_ = 6, &
785                        rsrc_ = 7, csrc_ = 8, lld_ = 9 )
786   double precision   ONE, ZERO
787   parameter( one = 1.0d+0, zero = 0.0d+0 )
788   ! ..
789   ! .. Local Scalars ..
790   integer            ictxt, mycol, myrow, npcol, nprow, i, j
791   double precision   scale_factor, elem
792   ! ..
793   ! .. External Subroutines ..
794   external           blacs_gridinfo, dlarnv, pdelset
795   ! ..
796   ! .. Executable Statements ..
797
798   !
799   ! Get grid parameters.
800   !
801   ictxt = desca( ctxt_ )
802   call blacs_gridinfo( ictxt, nprow, npcol, myrow, mycol )
803
804   !
805   ! Check matrix_type.
806   !
807   if( matrix_type == 1 ) then
808     !
809     ! Fill A with random numbers.
810     !
811     do j = 1, n
812       do i = 1, m
813         call dlarnv( 1, iseed, 1, elem )
814         call pdelset( a, i, j, desca, elem )
815       end do
816     end do
817
818   else
819     !
820     ! Fill A with integer entries converted to double precision, and
821     ! scaled down.
822     !
823     if( ( m == 0 ).or.( n == 0 ) ) then
824       scale_factor = one
825     else
```

```
826       scale_factor = one / ( dble( m ) * dble( n ) )
827     end if
828     elem = zero
829     do j = 1, n
830       do i = j, m
831         elem = elem + one
832         call pdelset( a, i, j, desca, elem * scale_factor )
833       end do
834       do i = 1, j - 1
835         elem = elem + one
836         call pdelset( a, i, j, desca, elem * scale_factor )
837       end do
838     end do
839     !
840     if( ( m > 0 ).and.( n > 0 ) ) then
841       elem = one + 0.1
842       call pdelset( a, 1, 1, desca, elem * scale_factor )
843     end if
844
845   end if
846   !
847   return
848   !
849   ! End of generate_random_matrix
850   !
```

Here is the caller graph for this function:



3.2.1.4 pd_test_and_time_randutv()

```
subroutine pd_test_and_time_randutv (
             integer ictxt,
             integer m,
             integer n,
             integer nb,
             integer q_factor,
             integer, dimension( * ) iseed,
             integer print_matrices,
             double precision wtime,
             double precision resid )
```

Tests and times the randUTV factorization.

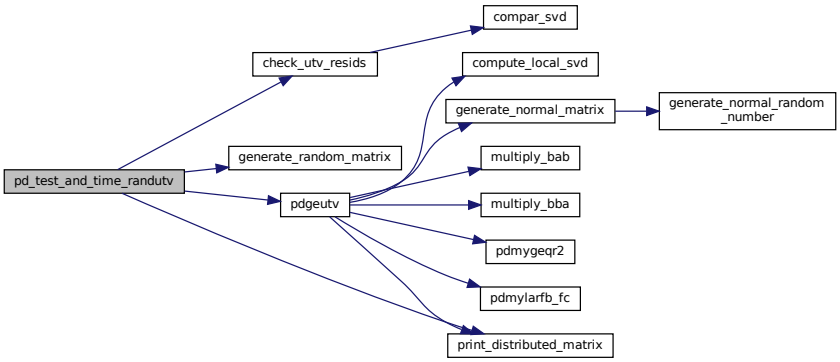
Definition at line 157 of file pdttrandutv.f90.

```
157   !
158   implicit none
159   !
160   ! .. Scalar Arguments ..
161   integer            ictxt, q_factor, m, n, nb, print_matrices
162   double precision   wtime, resid
163   ! ..
164   ! .. Array Arguments ..
165   integer            iseed( * )
166   ! ..
167   !
168   ! Purpose
169   ! =======
170   !
171   ! It tests and times the randUTV factorization.
```

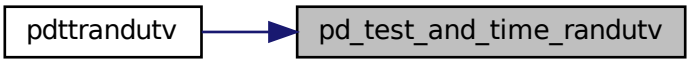
```
172   !
173   ! =====================================================================
174   !
175   ! .. Parameters ..
176   integer            BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DTYPE_, &
177                      LLD_, MB_, M_, NB_, N_, RSRC_
178   parameter( block_cyclic_2d = 1, dlen_ = 9, dtype_ = 1, &
179                        ctxt_ = 2, m_ = 3, n_ = 4, mb_ = 5, nb_ = 6, &
180                        rsrc_ = 7, csrc_ = 8, lld_ = 9 )
181   double precision   ONE, ZERO, TWO, BILLION
182   parameter( one = 1.0d+0, zero = 0.0d+0, two = 2.0d+0, &
183                        billion = 1.0d+9 )
184   ! ..
185   ! .. Local Arrays ..
186   integer            desca( DLEN_ ), descac( DLEN_ ), &
187                      descu( DLEN_ ), descv( DLEN_ ), &
188                      niseed( 4 )
189   double precision, dimension ( : ), allocatable :: a, ac, u, v
190   ! ..
191   ! .. Local Scalars ..
192   integer            nprow, npcol, myrow, mycol, &
193                      npa, nqa, npac, nqac, npu, nqu, npv, nqv, &
194                      j, info, allocstat
195   ! ..
196   ! .. External Functions ..
197   integer            numroc
198   double precision   check_utv_resids
199   external           numroc, check_utv_resids
200   ! ..
201   ! .. External Subroutines ..
202   external           blacs_barrier, blacs_gridinfo, descinit, &
203                      generate_random_matrix, pdgeutv, pdlacpy, &
204                      print_distributed_matrix, slboot, slcombine, sltimer
205   ! ..
206   ! .. Intrinsic Functions ..
207   intrinsic          int, max, dble
208   ! ..
209   ! .. Executable Statements ..
210
211   !
212   ! Get grid parameters.
213   !
214   call blacs_gridinfo( ictxt, nprow, npcol, myrow, mycol )
215
216   !
217   ! Some initializations.
218   !
219   npa   = numroc( m, nb, myrow, 0, nprow )
220   nqa   = numroc( n, nb, mycol, 0, npcol )
221   npac  = numroc( m, nb, myrow, 0, nprow )
222   nqac  = numroc( n, nb, mycol, 0, npcol )
223   npu   = numroc( m, nb, myrow, 0, nprow )
224   nqu   = numroc( m, nb, mycol, 0, npcol )
225   npv   = numroc( n, nb, myrow, 0, nprow )
226   nqv   = numroc( n, nb, mycol, 0, npcol )
227
228   !
229   ! Initialize array descriptors for the matrices.
230   !
231   call descinit( desca,  m, n, nb, nb, 0, 0, ictxt, max( 1, npa  ), info )
232   call descinit( descac, m, n, nb, nb, 0, 0, ictxt, max( 1, npac ), info )
233   call descinit( descu,  m, m, nb, nb, 0, 0, ictxt, max( 1, npu  ), info )
234   call descinit( descv,  n, n, nb, nb, 0, 0, ictxt, max( 1, npv  ), info )
235
236   !
237   ! Allocate local arrays.
238   !
239   allocate( a( npa * nqa ), stat = allocstat )
240   if ( allocstat /= 0 ) stop '*** Not enough memory for a ***'
241
242   allocate( ac( npac * nqac ), stat = allocstat )
243   if ( allocstat /= 0 ) stop '*** Not enough memory for ac ***'
244
245   allocate( u( npu * nqu ), stat = allocstat )
246   if ( allocstat /= 0 ) stop '*** Not enough memory for u ***'
247
248   allocate( v( npv * nqv ), stat = allocstat )
249   if ( allocstat /= 0 ) stop '*** Not enough memory for v ***'
250
251   !
252   ! Generate matrices A and AC.
253   !
254   do j = 1, 4
255     niseed( j ) = iseed( j )
256   end do
257   call generate_random_matrix( 1, m, n, niseed, a, desca )
258   call pdlacpy( 'all', m, n, a, 1, 1, desca, &
```

```
259                             ac, 1, 1, descac )
260
261   !
262   ! Print initial matrix A.
263   !
264   if( print_matrices == 1 ) then
265     call print_distributed_matrix( m, n, a, 1, 1, desca, 'ai' )
266   end if
267
268   !
269   ! Factorize matrix and generate orthonormal matrices U and V. Get times.
270   !
271   if( ( myrow == 0 ).and.( mycol == 0 ) ) then
272     write( *, * ) 'Starting pdgeutv...'
273   end if
274
275   call slboot()
276   call blacs_barrier( ictxt, 'all' )
277   call sltimer( 1 )
278
279   call pdgeutv( m, n, a, desca, 1, u, descu, 1, v, descv, &
280                 q_factor, info )
281
282   call sltimer( 1 )
283   call slcombine( ictxt, 'all', '>', 'w', 1, 1, wtime )
284
285   if( ( myrow == 0 ).and.( mycol == 0 ) ) then
286     write( *, * ) 'End of pdgeutv.'
287   end if
288
289   !
290   ! Check info argument returned.
291   !
292   if( info /= 0 ) then
293     if( ( myrow == 0 ).and.( mycol == 0 ) ) then
294       write( *, * )
295       write( *, * ) 'Info code returned by pdgeutv = ', info
296       write( *, * )
297     end if
298   end if
299
300   !
301   ! Print final matrices.
302   !
303   if( print_matrices == 1 ) then
304     call print_distributed_matrix( m, n, a, 1, 1, desca, 'af' )
305     call print_distributed_matrix( m, m, u, 1, 1, descu, 'uf' )
306     call print_distributed_matrix( n, n, v, 1, 1, descv, 'vf' )
307   end if
308
309   !
310   ! Check residuals.
311   !
312   if( info == 0 ) then
313     resid = check_utv_resids( m, n, ac, descac, u, descu, a, desca, v, descv )
314   else
315     resid = - one
316   end if
317
318   !
319   ! Print time.
320   !
321   if( ( myrow == 0 ).and.( mycol == 0 ) ) then
322     write( *, * )
323     write( *, '(1x,a,f12.5)' ) 'Time (in s.) spent in the factorization: ', &
324                                wtime
325   end if
326
327   !
328   ! Deallocate local arrays and vectors.
329   !
330   deallocate( a )
331   deallocate( ac )
332   deallocate( u )
333   deallocate( v )
334
335   return
336   !
337   ! End of pd_test_and_time_randutv
338   !
```

Here is the call graph for this function:



Here is the caller graph for this function:



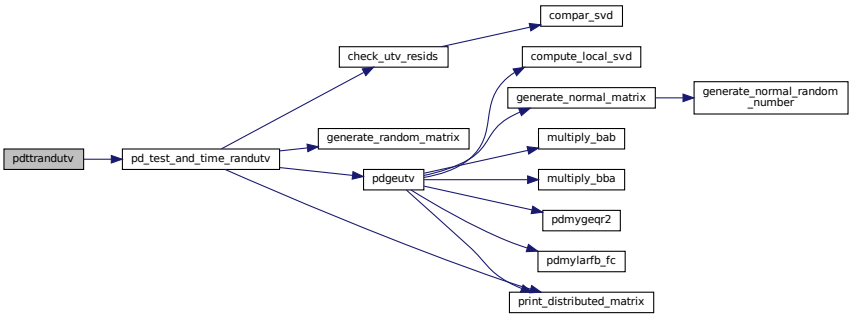
3.2.1.5 pdttrandutv()

`program pdttrandutv`

Main driver for randUTV computation.

Definition at line 2 of file pdttrandutv.f90.

Here is the call graph for this function:



3.2.1.6 print_distributed_matrix()

```
subroutine print_distributed_matrix (
             integer m,
             integer n,
             double precision, dimension( * ) a,
             integer ia,
             integer ja,
             integer, dimension( * ) desca,
             character*( * ) cname )
```

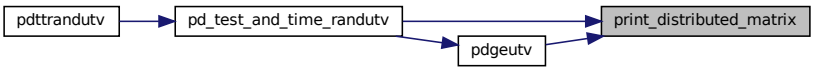
Prints the distributed matrix.

Definition at line 856 of file pdttrandutv.f90.

```
856   !
857   implicit none
858   !
859   ! .. Scalar Arguments ..
860   integer            m, n, ia, ja
861   character*( * )    cname
862   ! ..
863   ! .. Array Arguments ..
864   integer            desca( * )
865   double precision   a( * )
866   ! ..
867   !
868   ! Purpose
869   ! =======
870   !
871   ! It prints the distributed matrix.
872   !
873   ! =====================================================================
874   !
875   ! .. Parameters ..
876   integer            BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DTYPE_, &
877                      LLD_, MB_, M_, NB_, N_, RSRC_
878   parameter( block_cyclic_2d = 1, dlen_ = 9, dtype_ = 1, &
879                        ctxt_ = 2, m_ = 3, n_ = 4, mb_ = 5, nb_ = 6, &
880                        rsrc_ = 7, csrc_ = 8, lld_ = 9 )
881   double precision   ONE, ZERO
882   parameter( one = 1.0d+0, zero = 0.0d+0 )
883   ! ..
884   ! .. Local Scalars ..
885   integer            ictxt, myrow, mycol, nprow, npcol, &
886                      i, j
887   double precision   alpha
888   ! ..
889   ! .. External Subroutines ..
890   external           blacs_barrier, blacs_gridinfo, pdelget
891   ! ..
892   ! .. Executable Statements ..
893
894   !
895   ! Get grid parameters.
896   !
897   ictxt = desca( ctxt_ )
898   call blacs_gridinfo( ictxt, nprow, npcol, myrow, mycol )
899
900   ! Initial barrier.
901   call blacs_barrier( ictxt, 'All' )
902
903   ! Print matrix heading.
904   if( ( myrow == 0 ).and.( mycol == 0 ) ) then
905     write( *, fmt = '(a,a)' ) cname, ' = [ '
906   end if
907
908   ! Main loops.
909   do i = ia, ia + m - 1
910     do j = ja, ja + n - 1
911
912       ! Get and print element (i,j).
913       call pdelget( 'All', ' ', alpha, a, i, j, desca )
914       if( ( myrow == 0 ).and.( mycol == 0 ) ) then
915         write( *, fmt = '(1x,e14.7)', advance = 'no' ) alpha
916       end if
917
918     end do
919     ! Print end of line for every row.
920     if( ( myrow == 0 ).and.( mycol == 0 ) ) then
921       write ( *, * )
```

```
922     end if
923   end do
924
925   ! Print end of matrix.
926   if( ( myrow == 0 ).and.( mycol == 0 ) ) then
927     write( *, fmt = '(a)' ) '];'
928   end if
929
930   ! Final barrier.
931   call blacs_barrier( ictxt, 'All' )
932
933   return
934   !
935   ! End of print_distributed_matrix
936   !
```

Here is the caller graph for this function:



3.2.1.7 print_local_matrix()

```
subroutine print_local_matrix (
              integer m,
              integer n,
              double precision, dimension( lda, n ) a,
              integer lda,
              character*(*) cname )
```

Prints the local matrix in Matlab/Octave format.

Definition at line 942 of file pdttrandutv.f90.

```
942   !
943   implicit none
944   !
945   ! .. Scalar Arguments ..
946   integer            m, n, lda
947   character*(*)      cname
948   ! ..
949   ! .. Array Arguments ..
950   double precision   a( lda, n )
951   ! ..
952   !
953   ! Purpose
954   ! =======
955   !
956   ! It prints the local matrix in Matlab/Octave format.
957   !
958   ! =====================================================================
959   !
960   ! .. Local Scalars ..
961   integer            i, j
962   ! ..
963   ! .. Executable Statements ..
964
965   ! Print matrix heading.
966   write( *, fmt = '(a,a)' ) cname, ' = [ '
967
968   ! Main loops.
969   do i = 1, m
970     do j = 1, n
971         write( *, fmt = '(1x,e12.5)', advance = 'no' ) a( i, j )
972     end do
```

```
973       write ( *, * )
974     end do
975
976     ! Print end of matrix.
977     write( *, fmt = '(a)' ) '];'
978
979     return
980     !
981     ! End of print_local_matrix
982     !
```

3.3 README.md File Reference

Index

check_utv_resids
    pdttrandutv.f90, 25
compar_svd
    pdttrandutv.f90, 28
compute_local_svd
    pdgeutv.f90, 3

generate_normal_matrix
    pdgeutv.f90, 5
generate_normal_random_number
    pdgeutv.f90, 6
generate_random_matrix
    pdttrandutv.f90, 30

multiply_bab
    pdgeutv.f90, 7
multiply_bba
    pdgeutv.f90, 9

pd_test_and_time_randutv
    pdttrandutv.f90, 32
pdgeutv
    pdgeutv.f90, 11
pdgeutv.f90, 3
    compute_local_svd, 3
    generate_normal_matrix, 5
    generate_normal_random_number, 6
    multiply_bab, 7
    multiply_bba, 9
    pdgeutv, 11
    pdmygeqr2, 20
    pdmylarfb_fc, 21
pdmygeqr2
    pdgeutv.f90, 20
pdmylarfb_fc
    pdgeutv.f90, 21
pdttrandutv
    pdttrandutv.f90, 35
pdttrandutv.f90, 24
    check_utv_resids, 25
    compar_svd, 28
    generate_random_matrix, 30
    pd_test_and_time_randutv, 32
    pdttrandutv, 35
    print_distributed_matrix, 35
    print_local_matrix, 37
print_distributed_matrix
    pdttrandutv.f90, 35
print_local_matrix
    pdttrandutv.f90, 37