Conversion of a Sphere Optimization Program from LAPACK to ScaLAPACK

Paul Leopardi

The University of New South Wales,
School of Mathematics,
Department of Applied Mathematics

Draft only: 2002-12-03; Revised: TBD.

Abstract. The sphere optimization program, sphopt, was originally written as a sequential program using LAPACK, and was converted to use ScaLAPACK, primarily to overcome memory limitations. This document describes the overall design of sphopt in relation to the principles needed to convert LAPACK programs to ScaLAPACK, and gives details of this conversion. In particular, the ScaLAPACK version of sphopt uses a compressed block cyclic storage scheme to store two symmetric matrices in little more than the storage needed for one matrix.

1 Introduction

The sphere optimization program, sphopt, was originally written as a sequential program using LAPACK, and was converted to use ScaLAPACK, to overcome memory limitations, to speed processing, and to better use the resources provided by the Australian Partnership for Advanced Computing (APAC.)

1.1 The Problem to be Solved

The sphere optimization program, sphopt, uses one of a number of optimization methods to find a point set on the sphere which is optimal with respect to a certain function defined on point sets. The optimization follows a typical iterative structure:

repeat
  determine a search subspace;
  find the optimal point set in this subspace;
  update the current point set;
until the point set is locally optimal;

Finding the optimal point set within a subspace typically involves multiple function and gradient evaluations.
The function we are interested in here is the determinant of a Gram matrix. The Gram matrix, \( G \), is a function of the point set \( x \), defined by
\[
G_{i,j} := g(x_i \cdot x_j)
\]
where \( g \) is a Gegenbauer polynomial function.

Evaluation of the gradient needs the inverse of \( G \), and the value of the matrix \( DG \), the derivative of the Gram matrix:
\[
DG_{i,j} := g'(x_i \cdot x_j)
\]

1.2 The Need for ScaLAPACK

Typically, the Gram matrix is a large dense matrix. Its size depends on the size, \( m \), of the point set, which in turn depends on \( n \), the maximum polynomial degree for which the point set \( x \) is a fundamental system:
\[
m := (n + 1)^2
\]
For example, for degree \( n = 99 \), \( x \) consists of \( m = (n + 1)^2 = 10000 \) points, and \( G \) is a 10000 by 10000 dense matrix. For large degrees, the amount of storage needed for \( G \) and \( DG \) is too much for a typical uniprocessor system. This is why ScaLAPACK is needed.

2 Principles Needed for ScaLAPACK Parallelization

The organizing principles behind the ScaLAPACK parallelization of existing code containing LAPACK calls are those for ScaLAPACK code in general:

1. One program controls all processes
2. All processes run from the beginning of the program
3. ScaLAPACK and related calls imply some form of synchronization between processes

To elaborate on the first two points: the way a ScaLAPACK job usually runs is that a fixed number of processes run the same program. Processes are distinguished only by process id, and the program controls the processes by branching on the process id.

The third point needs a more elaborate explanation. You may want to also refer to the ScaLAPACK \([5]\) and BLACS \([2]\) documentation.
2.1 ScaLAPACK and Synchronization

ScaLAPACK and related calls are divided into three basic types:

1. **BLACS**: The Basic Linear Algebra Communications Subprograms organize communication between processes.
2. **PBLAS**: The Parallel Basic Linear Algebra Subroutines provide a parallel version of BLAS.
3. **ScaLAPACK proper**: The ScaLAPACK routines provide a parallel version of LAPACK.

When a ScaLAPACK or related call involves more than one process, it either creates or requires some form of synchronization between processes.

A good example of a call which creates synchronization is **DGEBR2D** [3], which is a broadcast receive.

“Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).” [3]

All ScaLAPACK proper routines, in contrast, require synchronization.

“All ScaLAPACK routines assume that the data has been distributed on the process grid prior to the invocation of the routine.” [4]

The result of all this is that to parallelize an existing program which uses LAPACK, it is necessary to examine each LAPACK or BLAS call, and any Fortran 90 operations on matrices, and convert each, if necessary, to use the following scheme:

1. Distribute operands
2. Synchronize
3. Operate
4. Distribute results

To parallelize an existing program which has LAPACK calls inside nested loops, it is almost certainly necessary to duplicate the structure of the nested loops so that all the participating processes execute the loops in parallel and stay in synch. For fixed loops, this is no problem, since the program will execute the same number of loops for all processes.

For varying loops, eg. loops governed by a varying termination condition, the situation is more complicated. In a ScaLAPACK program, the same data is not necessarily available to all processes unless it is explicitly made available. The safest way to implement varying loops in this case is to designate one process as the control process, and send it all data which determines the termination condition. The control process then broadcasts its decision to the other processes. This broadcast per loop keeps the processes in synch and ensures that all processes execute the same number of loops.
3 Structure of Sphopt

3.1 Overall Structure

The following pages of simplified Fortran pseudocode show the overall structure of both the sequential and parallel versions of sphopt.

PROGRAM sphopt
  Initialize
  DO trial = 1, ntrial
    ! Calculate log det on saved points before trial
    IF (scalapack == 0) THEN
      CALL ldw ! Sequential version
    ELSE
      ! Broadcast points from process 0,0 to all processes
      IF (myrow == 0 .AND. mycol == 0) THEN
        CALL DGEBS2D ! Broadcast send to all processes
      ELSE
        CALL DGEBR2D ! Broadcast receive from process 0,0
      END IF
      CALL pldw ! Parallel version
    END IF
    IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
      Set up random perturbation of points
    END IF
    ! Call appropriate optimization routine
    IF( INDEX('MD', spstr) /= 0) THEN
      IF (scalapack == 1) THEN
        Broadcast points from process 0,0 to all processes
      END IF
      IF (method == 5) THEN
        CALL lmbfgs ! See below
      ELSE
        CALL maxdet ! See below
      END IF
    END IF
    IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
      Reload best stored point set
    END IF
    Calculate log det on stored points after trial
    Calculate log det on current point set after trial
    IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
      Calculate update criteria and update point set, if necessary
    END IF
  END DO
  Finalize
END PROGRAM sphopt
SUBROUTINE lmbfgs
IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
  Initialize
END IF
IF (maxit == 0) THEN
  RETURN
END IF
DO
  IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
    call setulb ! Call the minimization routine
    Set itask to the appropriate value
  END IF
  IF (scalapack == 1) THEN
    Broadcast itask from process 0,0 to all processes
  END IF
  IF (itask == fg) THEN
    ! Calculate function and gradient
    IF (scalapack == 0) THEN
      CALL logdetfg ! Sequential version
    ELSE
      Broadcast points from process 0,0 to all processes
      CALL plogdetfg ! Parallel version
    END IF
  ELSEIF (itask == newx) THEN
    ! Decide whether to stop
    IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
      Calculate stopit from stopping criteria
    END IF
    IF (scalapack == 1) THEN
      Broadcast stopit from process 0,0 to all processes
    END IF
    IF (stopit == 1) THEN
      RETURN
    END IF
  ELSE
    RETURN
  END IF
END DO
END SUBROUTINE lmbfgs
SUBROUTINE maxdet
Initialize
IF (scalapack == 0) THEN
  CALL logdetfg ! Sequential version
ELSE
  CALL plogdetfg ! Parallel version
END IF
IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
  Initialize search direction, beta in CG method
END IF
DO it = 1, itmax ! Main loop
  IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
    Calculate stopit from stopping criteria
  END IF
  IF (scalapack == 1) THEN
    Broadcast stopit from process 0,0 to all processes
  END IF
  IF (STOPIT(1) == 1) EXIT
  IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
    IF (method == 1) THEN
      Calculate conjugate gradient search direction
    END IF
    Set up for line search
  END IF
Line search ! See below
  IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
    Calculate stopit from stopping criteria
  END IF
  IF (scalapack == 1) THEN
    Broadcast stopit from process 0,0 to all processes
  END IF
  IF (STOPIT(1) == 1) EXIT
  IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
    IF (method == 1) THEN
      Conjugate gradient update: Polyak-Ribiere
    END IF
    Update values, including new points
  END IF
END DO ! End main loop
Finalize
END SUBROUTINE maxdet
maxdet line search:
DO lsit = 1, lsmax
  IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
    xnew = x + step*d
  END IF
  ! Calculate function and gradient
  IF (scalapack == 0) THEN
    CALL logdetfg ! Sequential version
  ELSE
    Broadcast xnew from process 0,0 to all processes
    CALL plogdetfg ! Parallel version
  END IF
  IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
    Calculate variables related to step
  END IF
  IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
    Calculate stopit from stopping criteria
  END IF
  IF (scalapack == 1) THEN
    Broadcast stopit from process 0,0 to all processes
  END IF
  IF (STOPIT(1) == 1) EXIT
  IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
    Calculate step
  END IF
END DO

Distribution of Work to Processes The amount of work and storage needed to determine a search subspace and to perform the optimization is small in comparison to the amount needed to calculate the function and gradient. Therefore, in lmbfgs, the routine setulb is called only by process 0,0. Similarly, in maxdet, all work other than calculation of the function and gradient is done by process 0,0.

Control Variables, Broadcasting Data The variables scalapack, myrow and mycol are used for overall structure and control. scalapack is set to 0 if the program is running in sequential mode and set to 1 if the program is running in ScaLAPACK parallel mode. myrow and mycol are the process id. Process 0,0 is the control process.

If scalapack is 0, there is only one process and it must execute the sequential version of all routines. Otherwise, the program must broadcast relevant data from process 0,0 to all processes, before executing the parallel version. Each broadcast is globally blocking and acts as a synchronization point.

The following pseudocode snippet shows how these control variables are typically used.
Note that process 0,0 here broadcasts all points to all processes, even though any one process may need only some of the points for local computation. The matrices $G$ and $DG$ themselves need never be distributed.

! Calculate log det
IF (scalapack == 0) THEN
   CALL ldw ! Sequential version
ELSE
   ! Broadcast points from process 0,0 to all processes
   IF (myrow == 0 .AND. mycol == 0) THEN
      CALL DGEBS2D ! Broadcast send to all processes
   ELSE
      CALL DGEBR2D ! Broadcast receive from process 0,0
   END IF
   CALL pldw ! Parallel version
END IF

Broadcasting Variables to Stop Loops The variable stopit is used to control loop termination. Typically, the sequential version of the code will calculate stopit and use its value to determine when to terminate a loop. The parallel version will ensure that process 0,0 has all necessary information to calculate stopit, rely on process 0,0 to perform this calculation, broadcast stopit from process 0,0 to all processes, then use the value of stopit to determine when to terminate a loop.

The following pseudocode snippet illustrates this.

IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
   Calculate stopit from stopping criteria
END IF
IF (scalapack == 1) THEN
   Broadcast stopit from process 0,0 to all processes
END IF
IF (stopit == 1) THEN
   Exit loop
END IF

Broadcasting Variables to Control Execution Flow In subroutine lmbfgs, the variable itask is used to control flow of execution.

The sequential version of the code calls setulb which determines itask and uses its value to determine what to do next. In the parallel version, process 0,0 calls setulb which determines itask, then broadcasts itask to all processes. All processes then use the value of itask to determine what to do next.
The following pseudocode snippet illustrates this.

```plaintext
IF (scalapack == 0 .OR. (myrow == 0 .AND. mycol == 0)) THEN
    call setulb ! Call the minimization routine
    Set itask to the appropriate value
END IF
IF (scalapack == 1) THEN
    Broadcast itask from process 0,0 to all processes
END IF
IF (itask == fg) THEN
    Calculate function and gradient
ELSEIF (itask == newx) THEN
    Decide whether to stop
ELSE
    RETURN
END IF
```

4 Compressed Block Cyclic Storage of Symmetric Matrices

4.1 Requirements

We want to store two symmetric matrices, say $A$ and $B$, in block cyclic storage, such that:

1. the memory requirement is minimized, and
2. we can perform element-by-element operations on $A$ and $B$ locally.

To elaborate point 2, we want to store corresponding global blocks of $A$ and $B$ on the same process.

4.2 Block Cyclic Storage

The usual storage scheme for ScaLAPACK is called the Block Cyclic storage scheme [1] [5]. This uses two views of a matrix: the global view and the local (per process) view. Each of the two dimensions of a matrix (ie. rows, columns) is treated, separately, orthogonally, but in exactly the same way, so it is only necessary to describe one of the two dimensions, say rows.

In ScaLAPACK, the processes are also numbered using a two-dimensional scheme, with each process having a process row number and a process column number.

The concept of a block is common to the global and local views of a matrix. For a given matrix, there is a fixed row block size, ie. there is a fixed number of rows in a block, for all row blocks except possibly the last, which may be incomplete. Each row block in the global view is given a row block number and is allocated cyclically to a process row.
Eg. If there are \( n \) process rows, we have:

\begin{align*}
\text{ROW BLOCK:} & \quad 1 \ 2 \ \ldots \ n \ n+1 \ n+2 \ n+3 \ \ldots \ 2n \ 2n+1 \ \\
\text{PROCESS ROW:} & \quad 1 \ 2 \ \ldots \ n \ 1 \ 2 \ 3 \ \ldots \ n \ 1 \ \\
\text{ROW CYCLE:} & \quad 1 \ 1 \ \ldots \ 1 \ 2 \ 2 \ 2 \ \ldots \ 2 \ 3 \ \ldots
\end{align*}

In the local view of process row \( m \), we have:

\begin{align*}
\text{ROW CYCLE:} & \quad 1 \ 2 \ 3 \ \ldots \ k \\
\text{LOCAL ROW BLOCK:} & \quad 1 \ 2 \ 3 \ \ldots \ k \\
\text{GLOBAL ROW BLOCK:} & \quad m \ n+m \ 2n+m \ \ldots \ (k-1)n+m
\end{align*}

A more concrete example, with 8 global rows, 2 process rows and block size 3:

\begin{align*}
\text{GLOBAL ROW:} & \quad 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \\
\text{ROW BLOCK:} & \quad 1 \ 1 \ 1 \ 2 \ 2 \ 2 \ 3 \ 3 \\
\text{PROCESS ROW:} & \quad 1 \ 1 \ 1 \ 2 \ 2 \ 2 \ 1 \ 1 \\
\text{ROW CYCLE:} & \quad 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 2 \ 2 \\
\text{CYCLIC ROW:} & \quad 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 1 \ 2
\end{align*}

Local view of process row 1:

\begin{align*}
\text{GLOBAL ROW:} & \quad 1 \ 2 \ 3 \ 7 \ 8 \\
\text{GLOBAL ROW BLOCK:} & \quad 1 \ 1 \ 1 \ 3 \ 3 \\
\text{GLOBAL ROW CYCLE:} & \quad 1 \ 1 \ 1 \ 2 \ 2 \\
\text{GLOBAL CYCLIC ROW:} & \quad 1 \ 2 \ 3 \ 1 \ 2 \\
\text{LOCAL ROW BLOCK:} & \quad 1 \ 1 \ 1 \ 2 \ 2 \\
\text{LOCAL ROW:} & \quad 1 \ 2 \ 3 \ 4 \ 5
\end{align*}

Local view of process row 2:

\begin{align*}
\text{GLOBAL ROW:} & \quad 4 \ 5 \ 6 \\
\text{GLOBAL ROW BLOCK:} & \quad 2 \ 2 \ 2 \\
\text{GLOBAL ROW CYCLE:} & \quad 1 \ 1 \ 1 \\
\text{GLOBAL CYCLIC ROW:} & \quad 4 \ 5 \ 6 \\
\text{LOCAL ROW BLOCK:} & \quad 1 \ 1 \ 1 \\
\text{LOCAL ROW:} & \quad 1 \ 2 \ 3
\end{align*}

So, we have that each global row cycle corresponds exactly to a local row block for each process row, except possibly the last row cycle, which corresponds to a row block for the first few process rows.
4.3 Symmetric Matrix Scheme

In the following, we must assume that the row and column block sizes are equal.

We can now go back to considering the matrix as a 2D structure in terms of row and column cycles:

Cycle := (Row cycle, column cycle):

1,1 1,2 1,3 ...
2,1 2,2 2,3 ...
3,1 3,2 3,3 ...
...

For a symmetric matrix, in the global view, cycle \((m, n)\) equals cycle \((n, m)\) transposed. ScaLAPACK supplies routines which only examine the upper or lower triangle of a symmetric matrix, so, at the cycle level, we need only store the upper or lower triangle of cycles. Also, all cycles are square, except possibly the last cycle in a row or column.

Let’s choose the upper triangle. We can store the upper triangle of cycles of symmetric matrix \(A\) to the usual block cyclic storage scheme. This means that ScaLAPACK can operate directly on matrix \(A\). For matrix \(B\), we need to store three types of cycle:

1. Square cycles of the strict upper triangle of \(B\).
2. Square cycles of the diagonal of \(B\).
3. Non-square cycles of the ragged right hand edge of \(B\).

We can store cycles of type 1, say cycle \((m, n)\) with \(m < n\), in the lower triangle of \(A\), as cycle \((n, m)\). The most important thing to note here is that the cycle is stored as-is—it is not transposed. This is essential to ensure that we can perform element-by-element operations on \(A\) and \(B\) locally. Keeping the cycle as-is preserves the correspondences between the process row and the global cyclic row, and the process column and global cyclic column.

Cycles of type 2 and 3 must be stored in extra storage. Globally, for type 2, we need one cycle of columns and a full set of rows; for type 3, we need up to one cycle of columns and a full set of rows except for the last cycle, which overlaps with the last cycle of type 2. In both cases, we store the cycle as-is, not transposed.

With this type of storage scheme for matrices \(A\) and \(B\), we can perform element-by-element operations on \(A\) and \(B\) by remapping the global row and column indices of \(A\) and \(B\) to the local indices used by the storage scheme. For \(A\), this mapping is the usual block cyclic mapping for the upper triangle. For \(B\), the mapping uses local blocks corresponding to the three types of cycle listed above.

Note that while we can operate on \(A\) directly using ScaLAPACK routines, we cannot operate on \(B\) with ScaLAPACK while it is stored according to this scheme. We would need to copy the upper triangle of \(B\) to other storage to allow ScaLAPACK to work on \(B\).
4.4 Iteration and Indexing

The general organizing principle of iteration and indexing when converting code from LAPACK to ScaLAPACK is to iterate over global indexes and operate using local indexes. To accomplish this, it is necessary to use a number of mappings, which can be defined as functions, or more efficiently, as lookup tables.

``` FORTRAN
DO j = 2, m
   IF (gtoproc(j) == mycol) THEN
      lj = gtolocal(j)
      DO i = 1, j-1
         IF (gtoproc(i) == myrow) THEN
            li = gtolocal(i)
            Operate on local data using (li,lj)
         END IF
      END DO
   END IF
END DO
```

4.5 The Case of Sphopt

In the case of Sphopt, we want to store the matrices $G$ and $DG$ in minimal storage. We want to be able to use ScaLAPACK with $G$, so we store $G$ in the usual block cyclic scheme. We split the storage of $DG$ into three parts:

1. Strict upper triangle cycles of $DG$ in the strict lower triangle of $G$
2. Diagonal cycles of $DG$ in the local array DGDIAG
3. Ragged right hand edge cycles of $DG$ in the local array DGRAGG.

The following code snippet shows how the locally relevant parts of $G$ and $DG$ are calculated from the point set $X$ and stored in compressed block cyclic storage.

``` FORTRAN
DO j = 2, m
   IF (gtoproc(j) == mycol) THEN
      lj = gtolocal(j)
      bj = ltoblock(lj)
      libj = ltolib(lj)
      ! Calculate strict upper triangle elements in column j
      lim = 0
      DO i = 1, j-1
         IF (gtoproc(i) == myrow) THEN
            li = gtolocal(i)
            z(li) = X(1,i)*X(1,j) + X(2,i)*X(2,j) + X(3,i)*X(3,j)
            lim = lim + 1
         END IF
      END DO
      IF (lim > 0) THEN
```

``` FORTRAN
```

IF (lim > 0) THEN
CALL sphrkd(n, lim, z, gj, dgj, c1, c2, c3)
! Store elements and symmetric elements
DO i = 1, j-1
  IF (gtoproc(i) == myrow) THEN
    li = gtolocal(i)
    bi = ltoblock(li)
    libi = ltolib(li)
    G(li,lj) = gj(li)
    IF (bi == bj) THEN
      dgdiag(li, libj) = dgj(li)
    ELSEIF (bj*mb+libi > lm) THEN
      dgragg(li, libj) = dgj(li)
    ELSE
      G(bj*mb + libi, bi*mb + libj) = dgj(li)
    END IF
  END IF
END DO
END IF
END IF
END DO

The following code snippet shows element by element multiplication of G and DG.

DO i = 1, m
  IF (gtoproc(i) == myrow) THEN
    li = gtolocal(i)
    bi = ltoblock(li)
    libi = ltolib(li)
    DO j = i, m
      IF (gtoproc(j) == mycol) THEN
        lj = gtolocal(j)
        bj = ltoblock(lj)
        libj = ltolib(lj)
        IF (bi == bj) THEN
          dgij = dgdiag(li, libj)
        ELSEIF (bj*mb+libi > lm) THEN
          dgij = dgragg(li, libj)
        ELSE
          dgij = G(bj*mb + libi, bi*mb + libj)
        END IF
        G(li,lj) = G(li,lj) * dgij
      END IF
    END DO
  END IF
END DO
END IF
END DO
5 Initialization and Contexts

What follows is a commentary on a slightly simplified version of the initialization code in `sphopt`.

**ScaLAPACK** programs run as one program with multiple processes. The first step in initialization is to determine which is the current process (`iam`) and what is the total number of processes (`npall`).

```
CALL BLACS_PINFO(iam, npall)
```

**BLACS** uses system contexts to organize communication between processes. The next step gets the default system context (`ictxt`).

```
CALL BLACS_GET(0, 0, ictxt)
```

`sphopt` uses a square process grid, so the number of processes per row or column is the square root of `npall`. The next steps initialize the grid and obtain the current process row (`myrow`) and column (`mycol`).

```
nproc = FLOOR(SQRT(DBLE(npall)))
CALL BLACS_GRIDINIT(ictxt, '', nproc, nproc)
CALL BLACS_GRIDINFO(ictxt, nproc, nproc, myrow, mycol)
```

The compressed block cyclic storage scheme allocates enough space to accommodate the local number of rows (`locnrows`) and columns (`locncols`) corresponding to the current process row and column.

```
locnrows = NUMROC(m, mb, myrow, 0, nproc)
locncols = NUMROC(m, mb, mycol, 0, nproc)
ALLOCATE(G(locnrows,locncols))
ALLOCATE(DGDIAG(locnrows,mb))
ALLOCATE(DGRAGG(locnrows,mb))
```

`sphopt` needs a special context (`dctxt`) for diagonal processes. These processes are the only ones which have access to the diagonal of `G`. The initialization sets up a process map (`dmap`) and uses it to create `dctxt`.

```
ALLOCATE(dmap(nproc,1))
DO i = 1, nproc
  dmap(i,1) = BLACS_PNUM(ictxt, i-1, i-1)
END DO
CALL BLACS_GET(ictxt, 10, dctxt)
CALL BLACS_GRIDMAP(dctxt, dmap, nproc, nproc, 1)
IF (myrow /= mycol) THEN
  dctxt = -1
ENDIF
```

Finally, the initialization sets up the global to local and global to process lookup tables.

```
CALL setuplookups(m, nproc, mb, MAX(locnrows,locncols))
```
6 Calculating the Function Value and Derivative

What follows is a commentary on a slightly simplified version of the code in `plogdetfg` which is used to calculate the function value $f$ and gradient $df$. The commentary is given as a series of imperatives: actions to be performed.

Note the use of the following scheme:

1. Distribute operands
   - The routine `DESCINIT` is used to create the descriptors needed by the various `ScaLAPACK` routines.
2. Synchronize
   - The routine `BLACS_BARRIER` is used for synchronization, where necessary.
3. Operate
4. Distribute results
   - The routine `PDGEMR2D` is used to distribute results to process 0,0, where necessary.

6.1 Commentary on `Plogdetfg`

SUBROUTINE `plogdetfg`

Convert the point set $s$ from spherical coordinates to cartesian coordinates ($cx$.) Each process does this locally and so has a local copy of $cx$.

   n = nval
   CALL `s2x(ns, s, m, cx)`

Calculate the Gram matrix $G$ and derivative $DG$ from $cx$.

   CALL `pgramxd(n, m, lm, ln, cx)`

Use `PDLANSY` to calculate the 1-norm of $G$. The corresponding `LAPACK` call is `DLANSY` ([1] p136.)

   CALL `DESCINIT(descg, m, m, mb, mb, 0, 0, ictxt, lm, INFO)`
   CALL `BLACS_BARRIER(ictxt, 'All')`
   gnorm = `PDLANSY(norm, uplo, m, G, 1, 1, descg, work)`

Use `PDPOTRF` to calculate the Cholesky factorization of $G$. The corresponding `LAPACK` call is `DPOTRF` ([1] p25.)

   CALL `BLACS_BARRIER(ictxt, 'All')`
   CALL `PDPOTRF(uplo, m, G, 1, 1, descg, info)`

Save the logs of the diagonal elements.

This is another example of “iterate over a global index and operate using a local index.” In this case, the global index is $j$ and the local index is $lj$. The variable $lgj$ is the local part of the global vector used to store the logs of the diagonal elements of $G$. Use `PDGEMR2D` to distribute $lgj$ to the copy vector $clgj$ in process 0,0.
IF (myrow == mycol) THEN
DO j = 1, m
  IF (gtoproc(j) == mycol) THEN
    lj = gtolocal(j)
    lgj(lj) = log(G(lj,lj))
  END IF
END DO
END IF
CALL DESCINIT(desclgj, m, 1, mb, mb, 0, 0, dctxt, lm, INFO)
CALL DESCINIT(descclgj, m, 1, m, 1, 0, 0, dctxt, m, INFO)
! Distribute global lgj to process 0,0 copy clgj
CALL PDGEMR2D(m, 1, lgj, 1, 1, desclgj, clgj, 1, 1, descclgj, dctxt)
END IF

Use PDPOTRI to calculate the inverse of G. The corresponding LAPACK call is DPOTRI ([1] p25.)

CALL BLACS_BARRIER(ictxt, 'All')
CALL PDPOTRI(uplo, m, G, 1, 1, descg, info)

Use PDLANSY to calculate the 1-norm of G inverse.

CALL BLACS_BARRIER(ictxt, 'All')
ginorm = pdlansy(norm, uplo, m, G, 1, 1, descg, work)
gcond = gnorm*ginorm

Calculate integration weights. Use PDSYMV to multiply the symmetric matrix G by the vector ones, which contains all ones, giving the vector w. The corresponding BLAS call is DSYMV ([1] p142.)
This produces the sum of the columns of inv(G), since at this point, the global G contains the inverse of the original G. Use PDGEMR2D to distribute w to the copy vector cw in process 0,0.

CALL DESCINIT(desco, m, 1, mb, mb, 0, 0, ictxt, lm, INFO)
CALL DESCINIT(descw, m, 1, mb, mb, 0, 0, ictxt, lm, INFO)
CALL DESCINIT(desccw, m, 1, m, 1, 0, 0, ictxt, m, INFO)
CALL PDSYMV(uplo, m, 1d0, G, 1, 1, descg, &
  ones, 1, 1, desco, 1, Od0, w, 1, 1, descw, 1)
CALL PDGEMR2D(m, 1, w, 1, 1, descw, cw, 1, 1, descnw, ictxt)

Calculate function value f and weight statistics only in process 0,0.

IF (myrow == 0 .AND. mycol == 0) THEN
  Sum clgj to calculate function value f
  Use cw to calculate weight statistics wmin and wms
END IF

Set G to G.*DG by element by element multiplication. This sets G to inv(G).*DG with respect to the original value of G.

For code, see ‘‘The Case of Sphopt’’ above.
Create local part of global \( x \) from local copy \( cx \).

\[
\text{DO } j = 1, m \\
\text{IF (gtoproc(j) == mycol) THEN} \\
\quad x(:,gtolocal(j)) = cx(:,j) \\
\text{END IF} \\
\text{END DO}
\]

Calculate derivatives with respect to \( x \). Use \texttt{PDSYMM} to multiply the symmetric matrix \( G \) by the matrix \( cx \), giving the matrix \( dx \). The corresponding \texttt{BLAS} call is \texttt{DSYMM} ([1] p142.) Use \texttt{PDGEMR2D} to distribute \( dx \) to the copy vector \( cdx \) in process 0,0.

\[
\begin{align*}
\text{CALL DESCINIT(descx, 3, m, 3, mb, 0, 0, icxt, 3, INFO)} \\
\text{CALL DESCINIT(descdx, 3, m, 3, mb, 0, 0, icxt, 3, INFO)} \\
\text{CALL BLACS_BARRIER(icxt, 'All')} \\
\text{CALL PDSYMM('r', uplo, 3, m, 2D0, G, 1, 1, descg, & x, 1, 1, descx, 0D0, dx, 1, 1, descdx) } \\
\text{CALL DESCINIT(desccdx, 3, m, 3, m, 0, 0, icxt, 3, INFO)} \\
\text{CALL PDGEMR2D(3, m, dx, 1, 1, descdx, cdx, 1, 1, desccdx, icxt)}
\end{align*}
\]

Calculate gradient \( df \) only in process 0,0.

\[
\text{IF (myrow == 0 .AND. mycol == 0) THEN} \\
\quad \text{Use cdx to calculate gradient df} \\
\text{END IF}
\]

END SUBROUTINE plogdetfg

References

   \text{http://www.netlib.org/lapack/lug/index.html}

2. \textit{BLACS Routines}
   \text{http://www.netlib.org/blacs/BLACS/QRef.html}

   \text{http://www.netlib.org/blacs/BLACS/QRef.html#BR}

4. “Call the ScaLAPACK Routine” in [5]
   \text{http://www.netlib.org/scalapack/slug/node36.html}

   \text{http://www.netlib.org/scalapack/slug/index.html}