```
! HPF Multigrid program
! From a simple multigrid solver by John van Rosendale

        PROGRAM EXAMPLE

!       Declarations
        PARAMETER ( iblack = 0 )
        PARAMETER ( ired = 1 )
        PARAMETER ( nsize = 20000 )
        INTEGER   icount, iter, it,  nx,  ny
        REAL      error_max, error_value
        REAL, ALLOCATABLE :: u(:,:),  f(:,:),  r(:,:)
!HPF$   DISTRIBUTE u(BLOCK,BLOCK), f(BLOCK,BLOCK), r(BLOCK,BLOCK)

!       Read in important parameters
        WRITE(*, *) 'max iterations, max error, nx,  ny?'
        READ(*, *) iter,  error_max, nx,  ny

!       Allocate main arrays
!       Note: DISTRIBUTE declarations come into effect when the array is  allocated.
        ALLOCATE( u(0:nx,0:ny), f(0:nx,0:ny), r(0:nx,0:ny) )

!       Initialize scalars and arrays
        error_value = 1.0E9
        icount  = 0
        CALL RANDOM_SEED        ! Random number initialization intrinsic
        CALL RANDOM_NUMBER(u)   ! Random number generation intrinsic
        u = 10.0*u
        r = 0.0
        f = 0.0

!       Perform  iter  multigrid passes
        DO it = 1, iter, WHILE (error_value > error_max)

          CALL mg(u, f, nx, ny)
          CALL residual(r, u, f, 1, nx, ny)

          error_value = SQRT( SUM(r*r) )
          CALL scale(u, 1/error_value, nx, ny)

          WRITE(*, *)
          WRITE(*, *) 'iteration:', it, '  error redn', error_value

        END DO

        STOP
        END
```

```
! Multigrid solver subroutine
! Performs one pass of  V-cycle multigrid iteration  on the grids

        SUBROUTINE mg(u,f,nx,ny)
        PARAMETER (iblack = 0)
        PARAMETER (ired = 1)
        INTEGER   nx, ny
        REAL      u(0:nx,0:ny), f(0:nx,0:ny)
        INTEGER   jmp, jmp2, lev, ngrids, nxp, nyp
        REAL      r(0:nx,0:ny)
!HPF$   DISTRIBUTE u*(BLOCK,BLOCK), f*(BLOCK,BLOCK)
!HPF$   ALIGN r(:,:) WITH u(:,:)

        nxp = nx + 1; nyp = ny + 1; ngrids = ILEN( MIN(nxp,nyp)-1 )

!       SOR and restrict up to and including the coarsest grid

        DO lev = 1, ngrids-1
           jmp  = 2**(lev-1)
           jmp2 = 2*jmp
!          SOR ….best to do a red sweep last
           CALL smooth(u,f,iblack,jmp,nx,ny)
           CALL smooth(u,f,ired,jmp,nx,ny)
!          Restriction …. all grids stored in u array
           CALL residual(r,u,f,jmp,nx,ny)
           CALL rest(r,jmp,nx,ny)
           CALL save(r,f,jmp,nx,ny)
           f(0:nx:jmp2,0:ny:jmp2) = r(0:nx:jmp2,0:ny:jmp2)
           u(0:nx:jmp2,0:ny:jmp2) = 0.0
        END DO

!     SOR and prolong from coarsest to finest  grid

        DO lev = ngrids, 1, -1
           jmp2 = 2**(lev-1)
           jmp  = jmp2/2
!          SOR ….must do a black first to fix up points not set  by prolong!
           CALL smooth(u,f,iblack,jmp2,nx,ny)
           CALL smooth(u,f,ired,jmp2,nx,ny)
!          Prolongation
           IF (lev > 1) THEN
              CALL prolong(u,jmp,nx,ny)
              CALL unsave(f,r,jmp,nx,ny)
           END IF
        END IF

        RETURN
        END
```

```
! Residual subroutine -(Fortran 77 style)
!  Computes the residual on a grid (or subgrid)
        SUBROUTINE residual(r,u,f,jmp,nx,ny)
        INTEGER   jmp, nx, ny
        REAL      r(0:nx,0:ny), u(0:nx,0:ny), f(0:nx,0:ny)
        INTEGER   nxp, nyp, jmp2, ip, im, jp, jm
!HPF$   DISTRIBUTE r*(BLOCK,BLOCK), u*(BLOCK,BLOCK), f*(BLOCK,BLOCK)

        nxp = nx + 1
        nyp = ny + 1
        jmp2 = 2*jmp

        DO j = 0, ny, jmp
           jp = MOD(nyp + j + jmp, nyp)
           jm = MOD(nyp + j - jmp, nyp)
!          do left edge
           i = 0
           ip = jmp
           im = nxp - jmp
           r(i,j) = - u(i,j) + &
              0.25*(u(ip,j) + u(im,j) + u(i,jp) + u(i,jm)) + f(i,j)
!          do interior
           DO i = jmp, nxp-jmp2, jmp
              ip = i + jmp
              im = i - jmp
              r(i,j) = - u(i,j) + &
                 0.25*(u(ip,j) + u(im,j) + u(i,jp) + u(i,jm)) + f(i,j)
           END DO
!          do right edge
           i = nxp-jmp
           ip = 0
           im = nxp - jmp2
           r(i,j) = - u(i,j) + &
              0.25*(u(ip,j) + u(im,j) + u(i,jp) + u(i,jm)) + f(i,j)

        END DO

        RETURN
        END
```

```
! Successive Over-Relaxation subroutine
! Performs red or black pass of SOR iteration on a grid or  subgrid
        SUBROUTINE smooth(u,f,icolor,jmp,nx,ny)
        INTEGER   icolor, jmp, nx, ny
        REAL      u(0:nx,0:ny), f(0:nx,0:ny)
        INTEGER   nxp, nyp, nxloc, nyloc, jmp2, jp, jm, ip, im, i0
!HPF$   DISTRIBUTE u*(BLOCK,BLOCK), f*(BLOCK,BLOCK)

        nxp = nx + 1
        nyp = ny + 1
        jmp2 = 2*jmp
        nyloc = nyp/jmp
        nxloc = nxp/jmp

!       Perform a red or black SOR sweep

!HPF$   INDEPENDENT, NEW(jp,jm,i0,i,ip,im)
        DO j = 0, ny, jmp
          jp = MOD(nyp + j + jmp, nyp)
          jm = MOD(nyp + j - jmp, nyp)
          i0 = jmp*MOD(icolor + j/jmp, 2)
          DO i = i0, nx, jmp2
            ip = MOD(nxp + i + jmp, nxp)
            im = MOD(nxp + i - jmp, nxp)
            u(i,j) = 0.25*(u(ip,j) + u(im,j) + u(i,jp) + u(i,jm)) + &
              f(i,j)
           END DO
        END DO

        RETURN
        END
```

```
! restrict points from fine grid to coarse grid
        SUBROUTINE rest(r,jmp,nx,ny)
        INTEGER    jmp, nx, ny
        REAL       r(0:nx,0:ny)
        INTEGER    jmp2
!HPF$   DISTRIBUTE r*(BLOCK,BLOCK)

        jmp2 = 2*jmp
        r(0:nx:jmp2,0:ny:jmp2) = r(0:nx:jmp2,0:ny:jmp2) + &
             0.5*(r(0:nx:jmp2,jmp:ny:jmp2) + &
                  CSHIFT(r(0:nx:jmp2,jmp:ny:jmp2),-1,2) + &
                  r(jmp:nx:jmp2,0:ny:jmp2) + &
                  CSHIFT(r(jmp:nx:jmp2,0:ny:jmp2),-1,1)) + &
             0.25*(r(jmp:nx:jmp2,jmp:ny:jmp2) + &
                  CSHIFT(r(jmp:nx:jmp2,jmp:ny:jmp2),(/0,-1/)) + &
                  CSHIFT(r(jmp:nx:jmp2,jmp:ny:jmp2),(/-1,0/)) + &
                  CSHIFT(r(jmp:nx:jmp2,jmp:ny:jmp2),(/-1,-1/)))

        RETURN
        END


! prolong points from coarse to fine grid
        SUBROUTINE prolong(u,jmp,nx,ny)
        INTEGER    jmp, nx, ny
        REAL       u(0:nx,0:ny)
        INTEGER    jmp2
!HPF$   DISTRIBUTE u*(BLOCK,BLOCK)


!
!               compute only the following points
!               rest do not need to be done since they will be
!               updated with an sor sweep
!
        jmp2 = 2*jmp
        u(0:nx:jmp2,jmp:ny:jmp2) = u(0:nx:jmp2,jmp:ny:jmp2) + &
           0.5*u(0:nx:jmp2,jmp2:ny:jmp2)
        u(0:nx:jmp2,jmp:ny:jmp2) = u(0:nx:jmp2,jmp:ny:jmp2) + &
           0.5*CSHIFT(u(0:nx:jmp2,jmp2:ny:jmp2),1,2)
        u(jmp:nx:jmp2,0:ny:jmp2) = u(jmp:nx:jmp2,0:ny:jmp2) + &
           0.5*u(jmp2:nx:jmp2,0:ny:jmp2)
        u(jmp:nx:jmp2,0:ny:jmp2) = u(jmp:nx:jmp2,0:ny:jmp2) + &
           0.5*CSHIFT(u(jmp2:nx:jmp2,0:ny:jmp2),1,1)


        RETURN
        END
```

```
!     stores f at (black,black) points in neighboring (red,black)
!     r value
        SUBROUTINE save(r,f,jmp,nx,ny)
        INTEGER   jmp, nx, ny
        REAL      r(0:nx,0:ny),f(0:nx,0:ny)
        INTEGER   nxp, jmp2
!HPF$   DISTRIBUTE r*(BLOCK,BLOCK), f*(BLOCK,BLOCK)
        nxp = nx + 1
        jmp2 = 2*jmp
        FORALL (j = 0:ny:jmp2)
          FORALL (i = jmp2:nx:jmp2) r(i-jmp,j) = f(i,j)
          r(nxp-jmp, j) = f(0,j)
!          for safety
           FORALL (i = 0:nx:jmp2) f(i,j) = 0.0
        END FORALL
        RETURN
        END

! copy f value, stored in (red,black) r location, back to
! (black,black) where it belongs...
        SUBROUTINE unsave(f,r,jmp,nx,ny)
        INTEGER   jmp, nx, ny
        REAL      r(0:nx,0:ny),f(0:nx,0:ny)
        INTEGER   nxp, jmp2
!HPF$   DISTRIBUTE r*(BLOCK,BLOCK), f*(BLOCK,BLOCK)
        nxp = nx + 1
        jmp2 = 2*jmp
        FORALL (j = 0:ny:jmp2)
          FORALL (i = jmp2:nx:jmp2) f(i,j) = r(i-jmp,j)
          f(0,j) = r(nxp-jmp, j)
!          for safety
           FORALL (i = jmp:nx:jmp2) r(i,j) = 0.0
        END FORALL
        RETURN
        END

! scales u by fact
        SUBROUTINE scale(u,fact,nx,ny)
        INTEGER   nx, ny
        REAL    fact
        REAL    u(0:nx,0:ny)
        REAL    avg
        avg = SUM(u) / SIZE(u)
        u = fact*(u-avg)
        RETURN
        END
```

```
        SUBROUTINE print(u,nx,ny)
        INTEGER   nx, ny
        REAL    u(0:nx,0:ny)
!HPF$   DISTRIBUTE u*(BLOCK,BLOCK)
        WRITE(*,*)
        DO j = ny,0,-1
          WRITE(*,10) (u(i,j),i = 0,nx)
        END DO
        WRITE(*,*)
10      FORMAT(' ',12f7.3)
        RETURN
        END



        SUBROUTINE printj(u,jmp,nx,ny)
        INTEGER   nx, ny
        REAL    u(0:nx,0:ny)
!HPF$   DISTRIBUTE u*(BLOCK,BLOCK)
        INTEGER   nxp, nyp
        nxp = nx + 1
        nyp = ny + 1
        WRITE(*,*)
        DO j = nyp-jmp,0,-jmp
          WRITE(*,10) (u(i,j),i = 0,nx,jmp)
        END DO
        WRITE(*,*)
10      FORMAT(' ',12f7.3)
        RETURN
        END
```

```
!  Alternate form of Residual subroutine (Array assignment style)
        SUBROUTINE residual(r,u,f,jmp,nx,ny)
        INTEGER   jmp, nx, ny
        REAL    r(0:nx,0:ny), u(0:nx,0:ny), f(0:nx,0:ny)
!HPF$   DISTRIBUTE r*(BLOCK,BLOCK), u*(BLOCK,BLOCK), f*(BLOCK,BLOCK)
        INTEGER   nxp, nyp, jmp2, ip, im, jp, jm

        nxp = nx + 1
        nyp = ny + 1
        jmp2 = 2*jmp

        r(0:nx:jmp,0:ny:jmp) = - u(0:nx:jmp,0:ny:jmp) + &
           0.25*(CSHIFT(u(0:nx:jmp,0:ny:jmp),1,1) + &
                CSHIFT(u(0:nx:jmp,0:ny:jmp),-1,1) + &
                CSHIFT(u(0:nx:jmp,0:ny:jmp),1,2) + &
                CSHIFT(u(0:nx:jmp,0:ny:jmp),-1,2)) + &
           f(0:nx:jmp,0:ny:jmp)
        RETURN
        END
```

```
!Alternate form of Residual Subroutine
! This form must be called as
!       CALL residual(r(0:nx:jmp,0:ny:jmp),u(0:nx:jmp,0:ny:jmp), &
!          f(0:nx:jmp,0:ny:jmp))
! and the interface to the procedure ust be explicit at the point of call (i.e. either MODULE
! or INTERFACE must be used, or all fumctions must be in the same file).
        SUBROUTINE residual(r,u,f)
        INTEGER   jmp, nx, ny
        REAL    r(:,:), u(:,:), f(:,:)
!HPF$   INHERIT r, u, f

        r = - u + 0.25*(CSHIFT(u,1,1)+CSHIFT(u,-1,1)+&
           CSHIFT(u,1,2)+CSHIFT(u,-1,2)) + f
        RETURN
        END
```

```
! Alternate form of Residual subroutine -(FORALLstyle)
        SUBROUTINE residual(r,u,f,jmp,nx,ny)
        INTEGER   jmp, nx, ny
        REAL    r(0:nx,0:ny), u(0:nx,0:ny), f(0:nx,0:ny)
!HPF$   DISTRIBUTE r*(BLOCK,BLOCK), u*(BLOCK,BLOCK), f*(BLOCK,BLOCK)
        INTEGER   nxp, nyp, jmp2, ip, im, jp, jm

        nxp = nx + 1
        nyp = ny + 1
        jmp2 = 2*jmp
!       do interior
        FORALL ( i=jmp:nxp-jmp2:jmp, j=jmp:nyp-jmp2:jmp ) &
           r(i,j) = -u(i,j)+0.25*(u(ip,j)+u(im,j)+u(i,jp)+u(i,jm))+f(i,j)
!       do horizontal edges
        j = nyp-jmp
        FORALL ( i=jmp:nxp-jmp2:jmp )
!          do top: j = 0
           r(i,0) = -u(i,0) + &
              0.25*(u(i+jmp,0)+u(i-jmp,0)+u(i,jmp)+u(i,nyp-jmp)) + f(i,j)
!          do bottom: j = nyp-jmp
           r(i,j) = -u(i,j) + &
              0.25*(u(i+jmp,j)+u(im-jmp,j)+u(i,0)+u(i,j-jmp2)) + f(i,j)
        END FORALL
!       do vertical edges
        i = nxp-jmp
        FORALL ( j=jmp:nyp-jmp2:jmp ) &
!          do left : i = 0
           r(0,j) = - u(0,j) + &
              0.25*(u(jmp,j)+u(nxp-jmp,j)+u(i,j+jmp)+u(i,j-jmp)) + f(i,j)
!          do right  i = nxp-jmp
           r(i,j) = - u(i,j) + &
              0.25*(u(0,j)+u(i-jmp,j)+u(i,jp+jmp)+u(i,j-jmp)) + f(i,j)
        END FORALL
!       do corners
        i = nxp-jmp
        j = nyp-jmp
        r(0,0) = - u(0,0) + &
           0.25*(u(jmp,0)+u(nxp-jmp,0)+u(0,jmp)+u(0,nyp-jmp)) + f(0,0)
        r(0,j) = - u(0,j) + &
           0.25*(u(jmp,j)+u(nxp-jmp,j)+u(0,0)+u(0,j-jmp)) + f(0,j)
        r(i,0) = - u(i,0) + &
           0.25*(u(0,0)+u(i-jmp,0)+u(i,jmp)+u(i,nyp-jmp)) + f(i,0)
        r(i,j) = - u(i,j) + &
           0.25*(u(0,j)+u(i-jmp,j)+u(i,0)+u(i,j-jmp)) + f(i,j)
        RETURN
        END
```

```
! Alternate form, of SOR using array syntax
        SUBROUTINE smooth(u,f,icolor,jmp,nx,ny)
        INTEGER   icolor, jmp, nx, ny
        REAL    u(0:nx,0:ny), f(0:nx,0:ny)
!HPF$   DISTRIBUTE u*(BLOCK,BLOCK), f*(BLOCK,BLOCK)
        INTEGER   jmp2, i0, i1

        jmp2 = 2*jmp
        i0 = jmp*MOD(icolor,2)
        i1 = jmp*MOD(icolor+1,2)

!       Shift elements horizontally
        u(0:nx:jmp2,i0:ny:jmp2) = f(0:nx:jmp2,i0:ny:jmp2) + &
           0.25*(u(jmp:nx:jmp2,i0:ny:jmp2) + &
                CSHIFT(u(jmp:nx:jmp2,i0:ny:jmp2),-1,1))
        u(jmp:nx:jmp2,i1:ny:jmp2) = f(jmp:nx:jmp2,i1:ny:jmp2) + &
           0.25*(CSHIFT(u(0:nx:jmp2,i1:ny:jmp2),1,1) + &
                u(jmp:nx:jmp2,i1:ny:jmp2))
!       Shift elements vertically
        u(i0:nx:jmp2,0:ny:jmp2) = u(i0:nx:jmp2,0:ny:jmp2) + &
           0.25*(u(i0:nx:jmp2,jmp:ny:jmp2) + &
                CSHIFT(u(i0:nx:jmp2,jmp:ny:jmp2),-1,2))
        u(i1:nx:jmp2,jmp:ny:jmp2) = u(i1:nx:jmp2,jmp:ny:jmp2) + &
           0.25*(CSHIFT(u(i1:nx:jmp2,0:ny:jmp2),1,2) + &
                u(i1:nx:jmp2,0:ny:jmp2))

        RETURN
        END
```

Organization 1:

This version stores all grids in the same array. The fine grid fills the entire array. he next coarser grid is stored in every other entry, and in general the level K grid is stored in every 2**K'th element. The entire array is passed to every subroutine, allowing precise declarations of the distribution.

Advantages:

Conceptually clear.
Compiler has full information at subroutine boundaries; likely to use fast linkage convention.
Compiling communication fairly straightforward.
Partitioning computation fairly straightforward.

Disadvantages:

Uses power-of-2 strides.
Hard to get subscripts right.
Unclear how to optimize coarsest grid levels.
Unclear how to manage communication buffer storage.

Array Declaration:

```
!       In main prgram
        REAL, ALLOCATABLE :: u(:),  f(:)
!HPF$   DISTRIBUTE u(BLOCK,BLOCK), f(BLOCK,BLOCK)
        READ(*, *) nx,  ny
        ALLOCATE( u(nx,ny), f(nx,ny) )
```

Subroutine linkage:

```
!       Calling SOR subroutine
        CALL smooth(u,f,iblack,jmp,nx,ny)
!       In SOR subroutine
        SUBROUTINE smooth(u,f,icolor,jmp,nx,ny)
        INTEGER   icolor, jmp, nx, ny
        REAL    u(nx,ny), f(nx,ny)
!HPF$   DISTRIBUTE u*(BLOCK,BLOCK), f*(BLOCK,BLOCK)
```

References within subroutine:

```
        u(i+jmp,j-jmp)
        u(1+jmp+jmp:nx:jmp*2,1:ny-jmp:jmp*2)
        CSHIFT(u(1+jmp:nx:jmp*2,1+jmp:ny:jmp*2),(/1,-1/))
```

Organization 2:

This version stores data as in organization 1. However, it passes sections of arrays rather than the whole array.

Advantages:

Probably more "natural" to Fortran 90 than passing full arrays.
Conceptually clear.
Subscripts relatively simple.

Disadvantages:

Uses power-of-2 strides (but hidden from the programmer).
Compiler has less information at procedure boundaries; may require more expensive linkage convention.
Unclear how to optimize coarsest grid levels.
Very unclear how to manage communications buffers.

Unknowns:

Compiler has less information at procedure call; effect on optimizations unclear.
Some runtime processing needed for communication; cost of this (and/or optimizations of it) needs research.

Array declaration:

```
!        In main prgram
         REAL, ALLOCATABLE :: u(:,:),  f(:,:)
!HPF$    DISTRIBUTE u(BLOCK,BLOCK), f(BLOCK,BLOCK)
         READ(*, *) nx,  ny
         ALLOCATE( u(nx,ny), f(nx,ny) )
```

Subroutine linkage:

```
!        Calling SOR subroutine
         CALL smooth(u(1:nx:jmp),f(1:ny:jmp),iblack)
!        In SOR subroutine
         SUBROUTINE smooth(u,f,icolor)
         INTEGER   icolor
         REAL    u(:,:), f(:,:)
!HPF$    INHERIT u, f
!HPF$    DISTRIBUTE u*(BLOCK,BLOCK), f*(BLOCK,BLOCK)
         nx = UBOUND(u,1)   ! Note: not the same nx as in caller
         ny = UBOUND(u,2)   ! Note: not the same ny as in caller
```

References within subroutine:

```
         u(i+1,j-1)
         u(2:nx,1:ny-1)
         CSHIFT(u,(/1,-1/))
```

Organization 3:

This organization creates a list of grids, passing one grid to each subroutine. Conceptually, this is precisely the way that multigrid is usually taught. It is equivalent in effect to the usual implementation strategy of dividing up a large memory block, but is done by the compiler.

Advantages:

Conceptually very clear.
Subscripts relatively simple.
Avoids power-of-2 strides (mostly).
Compilation of communication relatively straightforward
Partitioning of computation relatively straightforward.
DISTRIBUTE after ALLOCATE can be used to give different distributions at different grid levels.

Disadvantages:

Unclear how to manage communications buffers.
Inelegant array allocation in current HPF.

Unknowns:

Some research needed for prolongation/restriction operations.
Compiler has less information at procedure call; effect on optimizations unclear.
Some runtime processing needed for communication; cost of this (and/or optimizations of it) needs research.

Array Declarations:

```
        In main prgram
        TYPE grids
          REAL, POINTER :: u(:,:), f(:,:)
        END TYPE grids
        TYPE grids, ALLOCATABLE :: grid_list(:)
        REAL, POINTER :: utmp(:),  ftmp(:)
!HPF$   DISTRIBUTE utmp(BLOCK,BLOCK), ftmp(BLOCK,BLOCK)
        READ(*, *) ngrids
        ALLOCATE ( grid_list(ngrids) )
        DO i = 1, ngrids
          nx = 2**(ngrids-i+1)-1
          ny = 2**(ngrids-i+1)-1
          ALLOCATE( utmp(nx,ny), ftmp(nx,ny) )
          grid_list(i)%u => utmp
          grid_list(i)%f => ftmp
        END DO
```

Subroutine linkage:

```
!       Calling SOR subroutine from main routine
        CALL smooth(grid_list(i)%u,grid_list(i)%f,iblack)
```

```
!       Calling SOR subroutine from another subroutine
        CALL smooth(u,f,blk)
!       In SOR subroutine
        SUBROUTINE smooth(u,f,icolor)
        INTEGER   icolor
        REAL    u(:,:), f(:,:)
!HPF$   DISTRIBUTE u*(BLOCK,BLOCK), f*(BLOCK,BLOCK)
        nx = UBOUND(u,1)
        ny = UBOUND(u,2)
```

References within subroutine:

```
        u(i+1,j-1)
        u(2:nx,1:ny-1)
        CSHIFT(u,(/1,-1/))
```

Notes:

This organization allows different distributions to be used for different grids, partly solving the optimization problems mentioned in organization 1 and 2. However, this drastically reduces the compiler information available. More research is needed to discover how this affects compiler organization and optimizations.