PSWE 代码阅读笔记

刘群*

地球系统科学研究中心

2015 年 4 月 8 日

这次作业要阅读的 PSWE 代码是上次浅水波方程求解过程的并行版本，主要利用 MPI 实现的并行化。在利用 MPI 时，这个程序进行了很好的封装，将一些调用的接口进行统一化、模块化。下面具体介绍一下各个模块的功能。

1 程序的整体框架

这个程序主要实现了浅水波方程隐式求解的并行化。在 pswe.f90 中，主要是调用其他的模块进行必要的 MPI 并行的初始化工作，然后设定每个进程的初始条件和相关参数，调用 euler 子模块进行时间积分，并在积分前后由 master 进程输出相关质量和能量信息。其他的子模块分别实现了不同的功能，主要如下：

1. kinds_mod.f90 主要定义了所有程序中要用到的整数、实数、布尔型、字符串等类型的相关信息；
2. communicate.f90 这个模块主要是对 MPI 进行初始化操作，对通信域进行划分，还有就是定义 master 打印的统一接口；
3. distribution.f90 这个模块主要是分配每个进程的格点数；
4. broadcast.f90 这个模块主要是定义了不同类型数据的统一接口；
5. module_para.f90 这个模块主要定义了每个进程要用到的一些参数，主要包括进程数、网格信息等，通过广播的形式传到各个进程；
6. module_array.f90 这个模块主要是定义并分配了每个进程要用到的数组，还有计算能量和质量的函数；
7. global_reduction.f90 这个模块主要是定义了全局归约求和的接口和各种数据 gather 和 scatter 的程序；
8. boundary.f90 这个模块主要是进行进程间边界数据的传递；
9. module_io.f90 这个模块主要是将数据写入文件中，同时进行数据的检查；
10. cs.f90 这个模块主要是计算每个进程的一些必要的参数；
11. haurwitz.f90 这个模块提供整个方程的初始条件；
12. dif.f90 这个模块主要是计算几个差分，要用到几个三对角分解的程序；
13. euler.f90 这个模块主要是进行时间积分，主要的求解过程在这个模块里。
14. exit_mod.f90 这个模块主要是退出程序的几种模式。

由于对于串行算法已经进行了阅读，因此在这里我将主要对并行部分进行详细的介绍。

2 主要模块的介绍

2.1 communicate.f90

这个模块主要是进行 MPI 的初始化和通信域的划分，同时定义了 master 进程输出信息的统一的接口。

下面的子程序 *init_communicate* 是该程序中对 MPI 进行初始化的操作，得到了进程号 *my_task* 和进程数 *nprocs*。

```
subroutine init_communicate     ! MPI初始化子程序
    ! local variables
    integer (int_kind) :: ierr  ! MPI error flag
    call MPI_INIT(ierr)         ! MPI初始化
    master_task = 0             ! master 进程ID为0
    ! 得到进程ID
    call MPI_COMM_RANK (MPI_COMM_WORLD, my_task, ierr)
    ! 总进程数
    call MPI_COMM_SIZE (MPI_COMM_WORLD, nprocs, ierr)
    comm = MPI_COMM_WORLD ! comm 代替MPI_COMM_WORLD
    ! MPI_DBL 代替MPI_DOUBLE_PRECISION
    MPI_DBL = MPI_DOUBLE_PRECISION
end subroutine init_communicate
```

在代码中的子程序 *create_comm_group* 是对通信域进行划分，其中 $nproc\_x$ 是 x 方向的进程数，$nproc\_y$ 是 y 方向的进程数。如果总共有 20 格进程，$nproc\_x = 4$, $nproc\_y = 5$, 则有 $color = 0, 1, 2, 3, 4$, $key = 0, 1, 2, 3$, 即 x 方向一行为一个通信域，通信域内编号分别为 $0, 1, 2, 3$.

```
subroutine create_comm_group(nproc_x, nproc_y, ierr)
    ! create latitude communicate group
    integer (int_kind), intent(in) :: nproc_x, nproc_y
    ! MPI error flag
    integer (int_kind), intent(out):: ierr
    call MPI_COMM_SPLIT(comm, my_task/nproc_x,&
        mod(my_task, nproc_x),comm_group, ierr)
end subroutine create_comm_group
```

在这个模块中，还定义了三个子程序 *master_print_str*, *master_print_str_dbl*, *master_print_str_int*, 但是除了接收的数据类型不同外，其他完全相同，因此可以将这三个子程序封装成统一接口，类似于 C++ 的重载，这样在以后的调用中可以直接调用 *master_print_message* 即可。

```
interface master_print_message
    module procedure master_print_str
    module procedure master_print_str_dbl
    module procedure master_print_str_int
end interface
```

2.2 distribution.f90

distribution 模块是一个分配网格点的模块，它也需要用到通信的模块。在这个模块里，定义了一些参数，如下表所示。

程序中的变量	实际意义
$nproc\_x$	每个进程中 x 方向的进程数
$nproc\_y$	每个进程中 y 方向的进程数
$nloc\_x$	每个进程中 x 方向的格点数
$nloc\_y$	每个进程中 y 方向的格点数

*电子邮件: liu-q14@mails.tsinghua.edu.cn，学号: 2014211591

程序中的变量	实际意义
nprocs	总进程数
tmpx	每个进程中 x 方向的格点数
tmpy	每个进程中 y 方向的格点数
iproc	进程的二维坐标 i
jproc	进程的二维坐标 j
iglobal	进程所在网格块起始点在 x 方向的格点坐标
jglobal	进程所在网格块起始点在 y 方向的格点坐标

下面这个子程序 *create_distribution* 对 x 和 y 方向的进程进行了划分，并求出了每个进程中的格点数。

```
subroutine create_distribution(nprocs, nproc_x, &
                        nproc_y,nglob_x, nglob_y)

    use mpi
    integer (int_kind), intent(in) :: &
    nprocs, &
    nproc_x, nproc_y, &  ! number of processors in
    this distribution
    nglob_x, nglob_y     ! number of grids in x and y
    direction
    ! nglob_y多了1
    ! local parameter
    integer (int_kind) :: &
            i,j,k,  ierr, &
            tmpx,         &
            tmpy
    ! dist下标从0到nproc_x(y)+1,
    外面多了一圈的虚拟进程
    if (.not. allocated( dist) ) &
        allocate(dist(0:nproc_x+1, 0:nproc_y+1))
    dist(:,:) = MPI_PROC_NULL    ! 暂时赋为虚拟进程
    call  MPI_BARRIER(comm, ierr) ! 同步

    ! x方向每个进程里的格点数
    tmpx  = nglob_x/nproc_x
    ! y方向每个进程里的格点数
    tmpy  = (nglob_y-1)/nproc_y

    do i = 0, nproc_x-1
        do j = 0, nproc_y-1
            k = j*nproc_x+i
            dist(i+1, j+1) = k           ! 进程ID为k
            if (my_task == k ) then      !
                iproc = i    ! 进程k的二维坐标
                jproc = j
     ! 进程k所在网格块起始点在x方向的格点坐标
                iglobal = iproc*tmpx
     ! 进程k所在网格块在起始点y方向的格点坐标
                jglobal = jproc*tmpy +1
                ! x方向每个进程里的格点数
                nloc_x  = tmpx
                !y方向每个进程里的格点数
                if (jproc == nproc_y-1) then
                      nloc_y  =nglob_y-jglobal-1
                else
                     nloc_y = tmpy
                endif
            end if
        end do
    end do

    do j = 1, nproc_y
        dist(0, j) = dist(nproc_x,j)  ! 循环边界条件
        dist(nproc_x+1, j) = dist(1,j)
    end do
end subroutine  create_distribution
```

2.3 broadcast.f90

broadcast 是将数据进行广播的模块，其主要是定义了数据发布的统一接口，包括标量和数组。

```
 !标量广播的统一接口(整合了双精度和整型数据)
interface broadcast_scalar
    module procedure broadcast_scalar_dbl
    module procedure broadcast_scalar_int
end interface

! 2d双精度数组广播的接口
interface broadcast_array
    module procedure broadcast_array_dbl_2d
end interface
```

由于这两个类似，我们以广播数组为例进行介绍。

```
 subroutine broadcast_array_dbl_2d(array, root_pe)
    ! Broadcasts a dbl 2d array from one processor (
    root_pe)
    ! to all other processors. This is a specific
    ! instance of the generic broadcast\_array
    interface.
    use mpi
    ! !INPUT PARAMETERS:
    integer (int_kind), intent(in) :: &
       root_pe    ! processor number to broadcast from
    ! !INPUT/OUTPUT PARAMETERS:
    real (r8), dimension(:,:), intent(inout) :: &
        array              ! array to be broadcast
    ! local variables
    integer (int_kind) ::  &
        nelements,         &! size of array
        ierr                ! local MPI error flag
    nelements = size(array) ! array数组的长度

    ! 将双精度数组array从root_pe广播出去
    call MPI_BCAST(array, nelements, MPI_DBL, root_pe,
     MPI_COMM_WORLD, ierr)
    call MPI_BARRIER(MPI_COMM_WORLD, ierr) ! 同步
end subroutine broadcast_array_dbl_2d
```

2.4 exit_mod.f90

这个模块主要是退出程序的几种模式，包括正常退出和中断等，这样就可以在程序中调用进行必要的检测，以便出错时进行调试。

```
 sdbroutine exit_PSWE(exit_mode, exit_message)
    ......
    if (my_task == master_task) then
        select case(exit_mode)
            case(sigExit)
                write (6,'(a14)') 'PSWE exiting...'
            case(sigAbort)
                write (6,'(a15)') 'PSWE aborting...'
            case default
                write (6,'(a37)') 'PSWE exiting with
    unknown exit mode...'
        end select
        write (6,*) exit_message
    endif
    ......
end subroutine exit_PSWE
```

2.5 module_para.f90

这个模块主要定义了每个进程要用到的一些参数，主要包括进程数、网格信息等，通过广播的形式传到各个进程；其中 x 和 y 方向的进程数、网格的分辨率和积分的时间、步长等信息是通过 namelist 的方式从外部读入。这些参数由 master 进程读入后，通过广播的形式广播到其他进程。

```
 subroutine init_para    ! 初始化参数子程序
    ......
    ! 定义namelists 格式
    namelist /dist_nml/  nproc_x, nproc_y
    namelist /domain_nml/  p, q
    namelist /time_manager_nml/  tlo, t0, t1
    ......
    ! master 从namelist读入进程数、分辨率和时间等信息
    if (my_task == master_task) then
      open (nml_in, file=nml_filename, status='old',&
                                iostat=nml_error)
      if (nml_error /= 0) then  ! 判断是否打开成功
         nml_error = -1
      else
         nml_error = 1
      endif
      read(nml_in, nml=dist_nml,iostat=nml_error)
      read(nml_in, nml=domain_nml,iostat=nml_error)
      read(nml_in, nml=time_manager_nml,&
                                iostat=nml_error)
      close(nml_in)
   endif
   ! 从master将读namelist的状态广播给所有进程
   call broadcast_scalar(nml_error, master_task)
   if (nml_error /= 0) then ! 如果读取失败，则退出
      call exit_pswe(sigabort,'error reading pswe_in')
```

```
   endif

    ! 从master将namelist中的各种参数广播给所有进程
    ! 注意调用统一的interface
    call broadcast_scalar(nproc_x, master_task)
    call broadcast_scalar(nproc_y, master_task)
    call broadcast_scalar(p, master_task)
    ......
    call broadcast_scalar(t1, master_task)
    ......
end subroutine init_para
```

另外，还对读入的参数是否合理进行了检查，确保 x 方向和 y 方向进程数乘积为总进程数，且网格信息合理，否则退出.

```
 ! 确保x方向和y方向进程数乘积为总进程数，否则退出
if (nproc_x *nproc_y /= nprocs) then
    call exit_pswe(sigabort,'error computing resource'
    )
endif
! perform some basic checks on domain
if (p < 1 .or. q  < 1 ) then
    ! 如果domain size 不符合要求则中断退出
    !*** domain size zero or negative
    call exit_pswe(sigabort,'invalid domain: size < 1'
    )
endif
```

同时，让 ID 为 1 的进程向屏幕打印必要的信息，包括积分时间步长，进程数目和分辨率等。

2.6 boundary.f90

这个模块主要是进行进程间边界数据的传递，在传递时引入了一些虚拟进程，同时每个进程的网格周围都多出了一圈网格，用以存放周围网格传递过来的数据。

下面这个程序 *create_boundary* 是找出一个进程的东西南北四个相邻的进程，由于 dist 的大小为 $nproc\_x+2$ 和 $nproc\_y+2$，因而里面已经存在了虚拟进程 MPI_NULL_PROC，不必担心进程不存在的问题。

```
 subroutine create_boundary
    integer :: i,j, ierr
    do j=1,nproc_y
        do i=1,nproc_x
            if (dist(i,j)  == my_task) then
                e_proc = dist(i+1, j) ! 东侧进程
                w_proc = dist(i-1, j) ! 西侧进程
                n_proc = dist(i, j+1) ! 北侧进程
                s_proc = dist(i, j-1) ! 南侧进程
            endif
        end do
    end do
    ! create latitude comm group
    call create_comm_group(nproc_x, nproc_y, ierr)
end subroutine create_boundary
```

子程序 *boundary_2d_dbl* 是进程间数据传递的程序，采用了非阻塞的方式，首先将数据读入缓冲区，然后将数据发送，或者将边界数据接收到缓冲区，在将数据发送接收后然后将数据更新到 ARRAY 数组中。下面以从西向东传送数据为例进行介绍。

$$w\_proc \to dist(i,j) \to e\_proc$$

```
 subroutine boundary_2d_dbl(ARRAY)
    ......
    !== send to east, rcv from west ==
    ! 非阻塞发送和接收
    ! MPI_IRECV中w_proc是源进程号
    call MPI_IRECV(buf_ew_rcv(1), nloc_y, mpi_dbl,    &
        w_proc, 1, comm, rcv_request, ierr)
    ! 将要发送的数据写入缓冲区
    ! 如果东侧进程不是虚拟进程
    if (e_proc /= MPI_PROC_NULL) then
        do  j = 1, nloc_y
            buf_ew_snd(j)  = ARRAY(nloc_x, j)
        end do
    end if
    ! MPI_ISEND中e_proc是目的进程号
    call MPI_ISEND(buf_ew_snd(1), nloc_y, mpi_dbl, &
                e_proc, 1, comm, snd_request, ierr)
    ! 等待非阻塞发送完成
    call MPI_WAIT(rcv_request, rcv_status, ierr)
    ! 将接收到的数据放到ARRAY预先设置的多出的位置中
    if (w_proc /= MPI_PROC_NULL) then
        do  j = 1, nloc_y
            ARRAY(0, j) = buf_ew_rcv(j)
        end do
    end if
    call MPI_WAIT(snd_request, snd_status, ierr)
    ......
end subroutine boundary_2d_dbl
```

其他的传递过程与这个类似，在这里就再对它们进行具体介绍。

在这个模块中，还有一个 *update_latitude* 的模块，这个函数中需要两个数组，这两个数组大小均是正常数组大小的一半，主要用途是用在求解三对角矩阵的求解过程中。原因是在求解三对角矩阵时，需要分奇偶项进行讨论，因此需要用到这个函数将计算结果进行整合。

```
 subroutine  update_latitude(ARRAY1,ARRAY2)
    ......
    ! 注意这里数组的大小，
    ! 第一维只有进程内x方向网格数的一半
    real (r8), dimension(1:nloc_x/2,1:nloc_y), &
        intent(inout) ::  ARRAY1, ARRAY2
    ! array containing horizontal slab to update
    ......
    ! 传送数据，下面与update_boudary相同，省略
    call MPI_IRECV(buf_ew_rcv(1), nloc_y, mpi_dbl,&
         w_proc, 2, comm, snd_request, ierr)
    ......

end subroutine update_latitude
```

2.7 global_reduction.f90

这个模块定义了程序中要用到的一些归约求和函数，定义了求和的统一接口 *global_sum*，这些接口主要是计算某些变量的和。

```
 interface global_sum  ! 求和的统一接口
    module procedure global_sum_dbl
    module procedure global_sum_scalar_dbl
end interface
```

这里也定义了两个相反的函数 *lat_gather* 和 *lat_scatter*，其中 *lat_gather* 是将纬向进程，即将一个通信域中的数据收集到本通信域的 0 号进程中，*lat_scatter* 是将纬向进程，即将一个通信域中 0 号进程中的数据收集到本通信域的其它进程中. 下面仅以 *lat_gather* 为例进行介绍。

```
 ! 将纬向进程中的数据收集到X中，即将一个通信域
! 中的数据收集到本通信域的0号进程中
subroutine lat_gather(X, Y)
    use mpi
    ! !INPUT PARAMETERS:
    real (r8), dimension(1:nloc_x*p,1:nloc_y*nproc_x),
     intent(inout) :: X
    real (r8), dimension(0:nloc_x+1,0:nloc_y+1),
    intent(in) :: Y ! array to be summed
    real (r8), dimension(1:nloc_x,1:nloc_y)  :: YY !
    array to be summed
    integer (int_kind) ::      &
                          i,j,  &  ! local counters
                          ierr     ! MPI error flag
    YY(:,:) = Y(1:nloc_x, 1:nloc_y) ! 一个进程中的数据
    ! 将纬圈的数据收集到X中，每个数据大小都是nloc_x*
    nloc_y
    call MPI_GATHER(YY(1,1),nloc_x*nloc_y,mpi_dbl, X
    (1,1),nloc_x*nloc_y, mpi_dbl, 0 ,  comm_group,
    ierr)
end subroutine lat_gather
```

还有关于求极点纬圈和的子程序 *polar_sum*，介绍如下：

```
 ! 求极点数据的和
subroutine polar_sum(X, polar_sum_s, polar_sum_n)
    ! sum up the values at polar circle
    use mpi

    ! INPUT PARAMETERS:
```

```
    real (r8), dimension(0:nloc_x+1,0:nloc_y+1),
    intent(in) :: X ! array to be summed
    ! !OUTPUT PARAMETERS:
    real (r8) ::      &
        polar_sum_s, &      ! resulting global sum
        polar_sum_n    real (r8) :: local_sum ! sum of
     all local blocks
    integer (int_kind) :: &
            i,j,           & ! local counters
            ierr             ! MPI error flag

    local_sum = .0
    if (jproc == 0 ) then ! south polar
        do i =1, nloc_x
                local_sum = local_sum + X(i,1)
        end do
        !一个全局归约的操作，将计算结果分
        ! 发给所有进程
        call MPI_ALLREDUCE(local_sum, polar_sum_s, 1,&
                mpi_dbl, MPI_SUM, comm_group, ierr)
    endif
    ! 北极也类似，故省略
    local_sum = .0
    if (jproc == nproc_y-1 ) then ! north polar
        ......
    endif

end subroutine polar_sum
```

2.8 module_io.f90

这个模块主要是 io 模块，主要作用是将每个进程里的数据写入到一个单独的文件中，文件名用两个进程号来标识。同时，这里还有一个对数据进行检查的程序 $check\_nan(mat, str)$，如果某个数据绝对值特别大，则会输出相应的提示信息。

```
 subroutine mpi_print(outfile, mat)
    use mpi
    integer :: i,j,ierr
    real(r8) :: mat(0:nloc_x+1,0:nloc_y+1)
    character(len=5) :: s1,s2
    character(*) outfile
    write(s1,"(I4)") jproc  ! 将进程二维坐标写成字符串
    write(s2,"(I4)") iproc
    ! 利用字符串函数得到要打开的文件名
    ! filename=jproc_iproc
    open(12,file=trim(adjustl(outfile))//trim(adjustl(
    s1))//'_'//trim(adjustl(s2)))

    if (jproc /= 0 .and. jproc /= nproc_y -1) then
        do j = 1,nloc_y ! 只写真实的数据
            do i = 1,nloc_x
                write(12,*), mat(i,j)
            enddo
        enddo
    endif
    ! 将南极的数据写入文件
    if (jproc == 0 ) then
        ......
    endif
    ! 将北极的数据写入文件
    if (jproc == nproc_y-1 ) then
        ......
    endif
    close(12) ! 关闭文件

end subroutine
```

2.9 cs.f90

cs.f90 是用来计算公式中要用到的参数值。比如说所有纬度的正弦值和余弦值，计算一些系数的值，比如 $c_{11}(j) = \frac{1}{2acos\theta_j \Delta\lambda}$, $c_{12}(j) = \frac{1}{2acos\theta_j \Delta\theta}$, $c_{13}(j) = \frac{1}{4acos\theta_j \Delta\lambda}$, $c_{14}(j) = \frac{1}{4acos\theta_j \Delta\theta}$, $j = 1, 2, \cdots, n$. 还有与科氏参数相关的量，比如 $f_0 = 2\omega_0 \sin\theta$, $f_2 = \frac{\tan\theta}{a}$, $f^* = f_0 + u * f_2 = 2\omega_0 \sin\theta + \frac{\tan\theta}{a}$. 由于该模块与串行代码基本一致，因此就不做过多介绍，唯一的区别就是，在这个程序中求的是一个进程中要用的参数值，而不是所有的进程的值。还需要指出的是，在下面的代码中求纬度的方法值得注意，jglobal 为某个进程起始位置的 j 坐标，再加上在进程内的坐标即可得到该格点在全局的坐标。

```
 subroutine cs
    ......
    do j=0,nloc_y+1
        ! jglobal为某个进程起始位置的j坐标
        ai    = (j+jglobal)*deta-detb ! 计算纬度值
        c1(j) = dcos(ai)              ! 余弦值
        s1(j) = dsin(ai)              ! 弦值
    end do
    ......
end subroutine cs
```

2.10 haurwitz.f90

这个模块还是为整个模拟提供初始条件，与串行版本不同的是，这里程序也是仅仅计算一个进程内的初始场，并且在计算完后要更新一下边界的值。

2.11 dif.f90

dif 模块是计算其中的某些差分项的值，在这里我们需要注意的是这里也是求一个进程内的值，在求完后需要调用进程间数据传递的函数更新边界，与串行程序还有一点不同是，这里引入了新的函数来设置某些参数在极点处的值，即 $polar\_setval$.

```
 subroutine polar_setval(wu, vals, valn)
    ......  ! 引入各种模块，省略
    real(r8), dimension(0:nloc_x+1,0:nloc_y+1), intent
    (inout) :: wu
    real(r8), intent(in):: vals, valn
    integer :: i, j

    ! south polar 注意此时jproc=0
    if(jproc == 0) then
        do i=1,nloc_x
            wu(i,0)=vals ! 极点展成一圈
        enddo
    endif
    ! north polar
    if(jproc == nproc_y-1) then
        do i=1,nloc_x
            wu(i,nloc_y+1)=valn
        enddo
    endif
    return
end subroutine polar_setval
```

2.12 euler.f90

这个模块是用来计算时间方向的积分过程，这是比较耗费时间的部分。在这里，与串行版本不同的是这里的一些参数采用了二维坐标来表达，在串行程序里只用一维数组表示。还有就是这里将求解线性代数方程组的过程进行了封装，比如求常系数、LU 分解等，通过 Tri_diag 和 LU 子程序来实现。

奇数项:fp 对应的方程中 $\varphi_{i+2,j}$ 的系数,rf 对应的方程右端常数项,fm 对应的方程中 $\varphi_{i-2,j}$ 的系数，偶数项:gp 对应的方程中 $\varphi_{i+2,j}$ 的系数，rg 对应的方程右端常数项，gm 对应的方程中 $\varphi_{i-2,j}$ 的系数. 有一点需要指出的是，这里的所有系数相关的项全部变成了二维数组，可能与这是算每个进程的量有关。

```
 subroutine Tri_diag(ra, rh, th )
    ! 引入各种模块
    ......
    implicit none
    ! 定义各种数组和变量
    integer i,i1,i2,j,k,hn,iter,ierr
    real(r8),dimension(0:nloc_x+1,0:nloc_y+1), intent(
    in)     :: ra,rh
    real(r8),dimension(0:nloc_x+1,0:nloc_y+1), intent(
    inout)    :: th
    ! 特别要注意下面的数组的大小，这里由于要分奇偶项
    ! 因此x方向大小只是nloc_x/2
    real(r8),dimension(1:nloc_x/2,1:nloc_y) :: fm,fp,
    f0,gm,gp,g0,rf,rg
    real(r8),dimension(1:nloc_y) :: sendbuf,recvbuf
    integer(int_kind) :: snd_req,rcv_req
    integer:: stat(MPI_STATUS_SIZE)
    real(r8) :: ai,aj
```

```
    hn = nloc_x/2   ! nloc_x的一半

    ! grid divided into two groups.
    ! 省略了各种系数的求解过程，与串行程序基本类似
    ! 这里与串行程序相比，每个参数都变成了二维数组
    ......

    ! 调用三对角矩阵求解程序求解
    call tridiagonal_solver(f0(:,:),fm(:,:),fp(:,:),rf
    (:,:), hn, nloc_y, nproc_x)
    call tridiagonal_solver(g0(:,:),gm(:,:),gp(:,:),rg
    (:,:), hn, nloc_y, nproc_x)
    ! 将结果重新赋给th(公式中的phi)
    do j=1,nloc_y
        do i=1,hn
            i1=i*2-1
            i2=i1+1
            th(i1,j)=rf(i,j)
            th(i2,j)=rg(i,j)
        enddo
    enddo
end subroutine Tri_diag
```

下面对 euler 里面用到的 *tridiagonal_solve* 函数做一个简单的介绍，这个函数的参数的意义如下：a, b, c 分别表示要求解的循环三对角矩阵的对角的元素，r 是方程右端的常数项，LN 和 M 表示这些参数的维度，在这个程序里，我们需要注意的是它采取了将所有要发送的数据同一先写到缓冲区里，然后再统一进行发送，这样可以减少通信的时间，提高程序的效率。

```
 ! the equation should be like:
!       |b c     a|
!       |a b c     |
!       |  a b c   | * x = r
!       |    a b c|
!       |c     a b|
```

2.13 pswe.f90

这个程序类似于串行程序中的“主程序”，主要是对各种模块进行调用，控制整个计算的过程，打印必要的信息。

```
 program PSWE
    use kinds_mod ! 导入各种module
    ......
    use mpi, only: MPI_WTIME ! 仅使用mpi中的mpi_wtime
    implicit none
    ! 定义各种参数
    integer(int_kind) :: i, j, ierr
    ......
    integer  :: iws,nt,nw,iwr,tt ! working variables
    ! 调用初始化模块，初始化MPI
    call  init_communicate

    ! caculate mpi runtime
    mpi_time_start = MPI_WTIME() ! 记录开始时间

    call  master_print_message("Init Parameter") !
    master进程打印消息
    call  init_para       ! 调用子程序初始化必要的参数
    call  master_print_message("Init Array")
    call  init_array      ! 调用子程序初始化一些数组
    call  master_print_message("Create Boundary")
    call  create_boundary !
    调用子程序设置每个进程的边界

    tt = t0              !hy start time
    iwr=(t1-tt)/tlo      !hy total step
    tlp=t1-tt-tlo*iwr    !hy last timestep

    ! Take care of the case when last timestep is
    different
    if (tlp.gt.0) then
        iwr=iwr+1
    else
       tlp=tlo
    end if
    ! master进程打印积分时间信息
    if (my_task == master_task ) then
        ! 打印各种信息
        ......
    end if

    call  master_print_message("Compute parameters")
    call  cs ! 计算用到的参数

    ! Using Rossby-Haurwitz waves as initial condition
    call master_print_message('Haurwitz Wave')
    call haurwitz

    ! variables transform
    ! 计算一个进程内的参数转换
    wu(:,:) = 0.0
    wv(:,:) = 0.0
    do j=1,nloc_y
        do i=1,nloc_x
            ai=dsqrt(wh(i,j))
            wu(i,j)=u(i,j)*i
            wv(i,j)=v(i,j)*ai
        end do
    end do
   ! 计算初始的质量和能量
    tener0=inner(wu,wv,wh,wu,wv,wh)
    tmass0 = mass(wh)

    nt=23
    dt=tlo
    nw=0
    ! 主进程负责打印初始能量质量积分时间等相关信息
    if (my_task == master_task ) then
        ......
    end if

    do iws=1,iwr
        ......
        ! The time integration
        ! 调用Euler子程序进行时间积分过程
        call euler(dt,iter)

        ! 将参数重新转换回来
        do j=1,nloc_y
            do i=1,nloc_x
                ......
            end do
        end do
        ! 根据某些变量值确定何时打印相关信息
        if ((nyn.eq.1).or.(int(tt/43200)*43200.eq.tt))
     then
            tener=inner(wu,wv,wh,wu,wv,wh)
            tmass = mass(wh)

            if (my_task.eq.0 .and. nyn.eq.0) then
                ......
            endif
            ......
        endif
    end do

    ! print wh for GrADS
    call mpi_print('out/100d_wh', wh)

    ! Finallize
    call destroy_array          ! 清除数组
    call destroy_distribution   ! 清理分配的结果

    mpi_time_end = MPI_WTIME()  ! 记录结束时间，并打印
    call master_print_message(mpi_time_end -
    mpi_time_start, "RUNTIME :")
    ! 打印运行成功的消息并退出
    call exit_PSWE(0, "Program PSWE end successfully!"
    )
end program PSWE
```

3 总结

通过这个程序 PSWE 我们可以看到，我们可以将一些接口封装起来，这样我们在调用时就会比较方便。同时，模块化也是我们在编写程序时需要考虑的内容，这样可以使得程序更加简洁，同时提高程序的复用性。另外，并行程序的执行效率确实较串行程序有了很大的提高。因此，在今后的科研中，我们要逐步学会自己去编写并行程序。