Middle Level Classes in UPIC Framework

Ufield2d class

This middle level class provides support for uniformly partitioned distributed scalar and vector arrays. Uniformly partitioned means that each processor’s piece of the distributed array is the same size. As an example, consider a global array with grid nx, ny:

```
integer, parameter :: nx = 32, ny = 64
real, dimension(nx,ny) :: a
```

If there are nvp processors, then with a 1D partition of this data, each processor’s part of this array would be dimensioned as follows:

```
integer, parameter :: nvp = 8
real, dimension(nx,ny/nvp) :: f
```

We assume the number of grids is exactly divisible by the number of processors. With a 2D partition, we need to specify the number of processors in the x and y direction:

```
integer, parameter :: nvpx = 2, nvpy = 4
real, dimension(nx/nvpx,ny/nvpy) :: f2
```

A vector field has several components for each grid element, such as the 2D electric field:

```
real, dimension(2,nx/nvpx,ny/nvpy) :: e
```
These fields may also contain **guard cells**, which are replicated elements that appear on more than one processor for optimization. The number of guard cells depends on the **order of interpolation** used. For example, for linear interpolation there is one extra guard cell on the right:

```fortran
real, dimension(nx+1,ny/nvp+1) :: f
```

With a 1D partition, the extra guard cell on one processor contains the first data element on the next processor:
A 2D partition has guard cells in two dimensions:

The main idea is that one processor “owns” the data, the other processor has a temporary copy in order to avoid excessive exchange of data between processors.
In a 2D code, the 1D partition always has one coordinate which is **local** (not distributed). This is an important advantage in developing some parallel algorithms, such as parallel FFTs. In particular, for algorithms which are **separable**, we can start with a layout where the x coordinate is local

```fortran
real, dimension(nx,ny/nvp) :: f
```

and perform some operation which requires locality in x, such as an FFT or a finite difference operation in x. Then transpose the data so that the y coordinate is local,

```fortran
real, dimension(nx/nvp,ny) :: g
```

and perform some other operation which requires locality in y, such as an FFT in y.

The main purpose of this class is to provide functions to perform **transposes** of data and to replicate and add up **guard cells** for 1D distributed data.

The philosophy followed here is that the class encapsulates information about uniformly partitioned data, that is, provides a **descriptor of the data**, but does not contain the data itself. The data is contained in ordinary Fortran90 arrays. Uniformly partitioned data is needed for spectral methods, such as FFTs.
The data structure for this class is:

```plaintext
type ufield2d
  integer :: partition, layout, mshare, ifbc, inorder
  integer :: nd1, nvpx, nd1p, n1blok
  integer :: nd2, nvpy, nd2p, n2blok
  integer :: n2min
end type ufield2d
```

which encapsulates the following information:

- **partition values**: 0=none, 1=1d, 2=2d
- **layout values**: XLOCAL(xy)=0, YLOCAL(yx)= 1
- **mshare values**: 0=distributed memory, 1= shared memory
- **ifbc** = field boundary flag
- **inorder** = interpolation order, 1=LINEAR, 2=QUADRATIC
- **nd1, nd2** = size of global array data in each dimension
- **nvpx, nvpy** = number of processors in each dimension
- **nd1p, nd2p** = size of local array data in each dimension
- **n1blok, n2blok** = number of shared processors/dimension
- **n2min** = expansion ratio for no buffer transpose of data

There are also several global constants defined:

**Layout constants:**

- XLOCAL = 0, YLOCAL = 1

**Interpolation order constants:**

- LINEAR = 1, QUADRATIC = 1
**Constructors** for creating a descriptor for uniformly partitioned 2d data:

```fortran
subroutine new_ufield2d_rl(this,nx,ny,nvpy,mshare,ifbc,
                          layout,order)
subroutine new_ufield2d(this,nx,ny,nvpy,mshare,ifbc,
                          layout,order)
```

The main idea here is that a user of this class does not need to know the details of what is inside the ufield2d type. The user only has to know about the **constructors**, which fill in the information inside the type, and **methods**.

**Accessor functions** for obtaining information about how to dimension arrays described by ufield2d objects:

```fortran
subroutine get_dim_ufield2d_rl(this,nd,inorder)
subroutine get_dim_ufield2d_cx(this,nd,inorder)
```

Usage: call get_dim_ufield2d_cx(this,nd) ! get dimension
allocate(sft(nd(1),nd(2),nd(3)))       ! allocate sft array

**Initialization functions** to assign values to arrays described by a ufield2d object:

```fortran
subroutine init_ufield(this,f,inorder,g) ! f = g
subroutine init_ufield(this,bxy,omx,omy,omz,inorder)
subroutine addto_ufield(this,f,that,g)   ! f = f + g
```

Main idea here is to make sure guard cells are handled properly.
**Transpose functions** to change data layouts:

```fortran
subroutine transp2d(source,f,dest,g)
subroutine transp2dv(source,f,dest,g)
subroutine transp2dx(source,f,dest,g)
subroutine transp2dxv(source,f,dest,g)
```

These transpose scalar and vector data. There are two versions which use different transpose algorithms, one of which uses less memory. They are used by the parallel fft programs, but can be used by any other algorithm which is separable. \( f(nx+1,ny/nvp+1) \Rightarrow g(ny+1,nx/nvp+1) \)

**Guard cell functions** for replicating data into guard cells:

```fortran
subroutine pcguardp(this,f,inorder)
subroutine pcguardp(this,f,inorder)
```

The two versions are for scalar and vector data. An example of their use is populating the guard cells of the electric field before pushing particles.

**Guard cell functions** for adding data from guard cells:

```fortran
subroutine paguardp(this,f,inorder)
subroutine paguardp(this,f,inorder)
```

The two versions are for scalar and vector data. An example of their use is adding the guard cells of the charge density to obtain total density.
Special functions used by **Dirichlet solver** to double the grid or extract half the grid.

- subroutine `dblslin(this,q,that,q2,inorder)`
- subroutine `hafdbl(this,fxy,that,fxy2,inorder)`
- subroutine `hafdbl(this,q,that,q2,inorder)`

To perform a **Fast Sine Transform**, we double the grid in an anti-symmetric fashion, perform an ordinary FFT, and keep only the sine terms. These functions are provided to support this.

This class is used by other classes which need to refer to uniformly partitioned fields, such as the `fft2d` class.
Here is the implementation of one of the functions:

    subroutine transp2d(source,f,dest,g)
    ! this subroutine performs transpose between different layouts
    ! source, dest = ufield2d descriptors of data
    ! f = source for transpose
    ! g = result of transpose
    ! kstrt = starting data block number, a global variable
    implicit none
    type (ufield2d), intent(in) :: source, dest
    complex, dimension(:,:,:) :: f, g
    ! local data
    integer :: nx, ny, nxv, nyv, kxp, kyp, kxpds, kypds
    integer :: jblok, kblok
    complex, dimension(dest%nd2p,source%nd2p,size(f,3)) :: s
    complex, dimension(dest%nd2p,source%nd2p,size(g,3)) :: t
    character(len=10), save :: sname = ':transp2d:'
    if (monitor==2) call werrfl(class//sname//' started')

    ! check for errors
    if (monitor > 0) then
      if (((source%partition /= 1) .or. (dest%partition /= 1)) then
        erstr = ' invalid / non-conforming partition'
        UFIELD2D_ERR = 6; EXCEPTION = EXCEPTION + 1
        call ehandler(EDEFAULT, class//sname// erstr); return
      endif
      if (((source%mshare /= 0) .or. (dest%mshare /= 0)) then
        erstr = ' non-conforming memory sharing'
        UFIELD2D_ERR = 7; EXCEPTION = EXCEPTION + 1
        call ehandler(EDEFAULT, class//sname// erstr); return
      endif
    endif
! obtain transpose arguments
  nxv = size(f,1); nyv = size(g,1)
  kxp = dest%nd2p; kyp = source%nd2p
  kxpd = size(g,2); kypd = size(f,2)
  jblok = dest%n2blok; kblok = source%n2blok
  nx = source%nd1; ny = source%nd2
  if (abs(source%layout - dest%layout)==1) then
    ! make sure arrays are conforming
    if ((source%nd1 /= dest%nd2) .or. (source%nd2 /= dest%nd1)) &
        &then
      erstr = ' non-conforming array'
      UFIELD2D_ERR = 8; EXCEPTION = EXCEPTION + 1
      call ehandler(EDEFAULT,class//sname//erstr); return
    endif
  ! perform transpose
    call PTPOSE(f,g,s,t,nx,ny,kstrt,nxv,nyv,kxp,kyp,kxpd,kypd,jblok,kblok)
  ! unsupported transpose
  else
    erstr = ' unsupported transpose'
    UFIELD2D_ERR = 9; EXCEPTION = EXCEPTION + 1
    call ehandler(EDEFAULT,class//sname//erstr); return
  endif
  if (monitor==2) call werrfl(class//sname//' complete')
end subroutine transp2d
Nfield2d class

This middle level class provides support for non-uniformly partitioned distributed scalar and vector arrays. Non-uniformly partitioned means that each processor’s piece of the distributed array is not the same size. An example of what non-uniformly partitioned data looks like is as follows:

1D Domain decomposition

2D Domain decomposition

Each partition has equal number of particles

Such data appears in PIC codes to ensure each processor has the same amount of work to perform.
The data structure for this class is:

```fortran
  type nfield2d
    integer :: partition, mshare, inorder, ifbc
    integer :: nx, ny
    integer :: nvpx, nvpy
    integer :: nxblok, nyblok
    real, dimension(:,,:), pointer :: edges
    integer, dimension(:,,:), pointer :: noff, nxyp
  end type nfield2d
```

which encapsulates the following information:

partition values: 0=none, 1=1d, 2=2d
mshare values: 0=distributed memory, 1=shared memory
inorder = interpolation order, 1=LINEAR, 2=QUADRATIC
ifbc = field boundary flag
nx, ny = size of global array data in each dimension
nvpx, nvpy = number of processors in each dimension
nxblok, nyblok = number of shared processors/dimension
edges(2*i-1,m) = left boundary in i direction in partition m
edges(2*i,m) = left boundary in i direction in partition m
noff(i,m) = smallest global gridpoint in i direction
  for linear 1d partition: noff(1,l) = int(edges(1,l))
nxyp(i,m) = number of gridpoints in i direction
  for linear: nxyp(1,l) = int(edges(2,l)) - noff(1,l)
for 1d partition, i = (1) refers to y
for 2d partition, i = (1,2) refers to (x,y)

Typical array size: f(nx,nxyp(1,l),nyblok=1)
This class describes arrays that also contains guard cells. The data layout for this class is always XLOCAL.

**Constructors** for creating a descriptor for non-uniformly partitioned 2d data:

subroutine new_nfield2d(this,spect)
subroutine new_nfield2d(this,rspace,order)

One constructor creates an nfield2d object from a spect2d object, the other creates a non-uniformly partitioned object which is actually uniformly-partitioned.

**Destructor:**

subroutine del_nfield2d(this)

**Accessor functions** to obtain area occupied by plasma. Used to normalize force to yield a plasma frequency of unity:

function get_area_n(this)

**Initialization function** to assign values to arrays described by an nfield2d object:

subroutine init_nfield2d_cv(this,bxy,omx,omy,omz)
subroutine sguardp(this,cu,xj0,yj0,zj0)
subroutine sguardp(this,q,qi0)
Guard cell functions for replicating data into guard cells:

    subroutine cguardp(this,fxy)
    subroutine cguardp(this,q)

Guard cell functions for adding data from guard cells:

    subroutine aguardp(this,cu)
    subroutine aguardp(this,q)

Functions are provided for both scalar and vector fields. Currently, the guard cells functions are implemented only in the non-distributed (XLOCAL) direction.

This class is used by others which require non-uniformly partitioned data, such as the 2d particle class.
FFT2d class

This middle level class provides functions to perform real to complex 2d FFTs for uniformly partitioned scalar or vector data.

The data in real space is required to have a layout where the x coordinate is local, and in fourier space where the y coordinate is local.

FFTs require the use of tables for storing trigonometric functions and bit reverse addresses. To avoid duplicating these tables, a table manager is created to give out pointers to tables, and will only create a new table if it is different than a previously created table.

The data structure for this class is:

type fft2d
   integer, dimension(2) :: ind
   integer, dimension(:), pointer :: mixup
   complex, dimension(:), pointer :: sct
end type fft2d

which encapsulates the following information:

ind = exponent which determines length in each direction
mixup = array of bit reversed addresses
sct = sine/cosine table
**Initialization functions** to obtain fft tables:

- function get_fft2table(indx, indy)
- subroutine del_fft2table(table)

A linked list of fft tables is maintained in an internal structure called table_list. When a request for an fft table is made, a search is first made to see if such a table has already been created. If so, the table is returned and the reference count for that table is incremented. If not, the table is created, and the reference count set to 1. When a table is requested to be deleted, the reference count is first decremented, but it is not actually deleted until the reference count is 0, since it is still in use.

**FFT functions** for 2d real to complex transforms:

- subroutine fft(this, rspace, f, kspace, g, isign, inorder)

There are FFTs for both scalar and vector data.
Fpois2d class

This middle level class provides functions to solve Poisson’s equation in fourier space for scalar or vector data. The data is required to be uniformly partitioned in fourier space where the y coordinate is local.

The poisson solvers require their own tables, and a table manager similar to the one in the FFT2d class is provided to avoid table duplication.

The data structure for this class is:

    type fpois2d
       integer, dimension(2) :: nd
       real, dimension(2) :: a
       real :: anorm
       integer :: n2blok, psolver
       complex, dimension(:,;,:), pointer :: ffc
    end type fpois2d

which encapsulates the following information:

nd = system length in each direction
a = half-width of particle in each direction
anorm = normalization constant for poisson solver
n2blok = number of shared processors in second dim.
psolver = type of poisson solver (defaults to periodic)
ffc = table for poisson solver
Initialization functions to obtain poisson tables:

function get_pois2table(kspace,ax,ay,affp,psolve)

subroutine del_pois2table(table)

Poisson solver functions for periodic or dirichlet boundary conditions (zero potential):

subroutine pois(this,q,fx,we)
subroutine pois(this,q,fxy,we)
subroutine bpois(this,cu,bxy,isign,we)

There are Poisson solvers for both scalar (such as potential) and vector (such as magnetic field) data.
Fdist2d class

This middle level class provides functions to describe distributions of particles. Distributions types are used to initialize particle co-ordinates. Currently, only bi-maxwellian velocity distributions in velocity and uniform distributions in space are supported. One important function provided is an estimate of how large particle arrays should be dimensioned.

The data structure for this class is:

```fortran
type fdist2d
  real :: vtx, vty, vtz
  real :: vdx, vdy, vdz
  integer :: npx, npy
  integer :: idimp, np
end type fdist2d
```

which encapsulates the following information:

- $vtx/vty/vtz =$ thermal velocity of particles in $x/y/z$
- $vdx/vdy/vdz =$ drift velocity of particles in $x/y/z$
- $npx/npy =$ number of particles distributed in $x/y$
- $idimp =$ dimension of phase space = 4 or 5
- $np =$ total number of particles
Constructors for creating a descriptor for 2d distributions:

subroutine new_fdist2d(this,vtx,vty,vdx,vyd,npx,npy)
subroutine new_fdist2d_h(this,vtx,vty,vtyz,vdx,vyd,vdz,
                      npx,npy)

Both 2D and 2-1/2 D distributions are supported.

Accessor functions for obtaining information about how to dimension particle arrays:

function size(this)
subroutine get_dim_fdist2d(this,nd,nvp,mshare)
subroutine get_dim_fdist2d(this,that,nd,nvp,mshare)
subroutine get_dim_fdist2d(np,nd,nvp,mshare)

This class is used by others which require 2d distributions, such as the 2d particle class.
Part2d class

This middle level class contains information about particle properties and information needed to process particles.

The data structure for this class is:

```fortran
  type part2d
    real :: qm, qbm, dt
    integer :: ipbc, emf, nbmax, np
    integer, dimension(:), pointer :: npp
    integer :: popt, dopt, djopt, vect, sortime
  end type part2d
```

which encapsulates the following information:

qm = charge on particle, in units of e  
qbm = particle charge/mass ratio  
dt = time interval between successive calculations  
ipbc = particle boundary flag  
emf = (0,1,2)=(electrostatic,magnetized es,magnetostatic)  
bymax = size of buffer for passing particles  
np = total number of particles  
npp(m) = number of particles in partition m  
popt = particle optimization flag  
dopt = charge deposit optimization flag  
djopt = current deposit optimization flag  
vect = (0,1) vector architecture mask  
sortime = number of time steps between particle sorting

There are also several global constants defined:
Algorithm constants:

STANDARD = 1, LOOKAHEAD = 2, VECTOR = 3

Electromagnetic constants:

ES = 0, MAGNETIZED_ES = 1, MSTATIC = 2, EM = 3

Constructors for creating a descriptor for 2d particles:

subroutine new_part2d(this,ipbc,emf,qm,qbm,dt,stime,
popt,dopt,djopt)

Destructors:

subroutine del_part2d(this)

Accessor function for obtaining information about total number of particles in simulation. Used to normalize force to yield a plasma frequency of unity:

function get_num(this)

Initialization functions for particle co-ordinates:

subroutine init_part(this,part,nspace,fdist)
subroutine init_gcenter(this,part,nspace,bxy,gcflag)
Functions for **depositing charge, current**:

  subroutine dpost(this,part,nspace,q)
  subroutine djpost(this,part,nspace,cu)

Functions for **advancing particles**:

  subroutine push(this,part,nspace,fxy,bxy,ek)

Functions for **sorting particles**:

  subroutine pmove(this,part,nspace)
  subroutine sortp(this,part,nspace)
Input2d class

This low level “class” defines the input namelist variables and provides default values for each of them. Namelists are a means for grouping sets of variables so that one can read and write them as a group, and are commonly used for input. The namelist is normally read only on node 0.

Two functions are provided to read the namelist variables and broadcast them to the other nodes.

General function to read 2d namelist variables:

```fortran
subroutine readnml2(name)
```

General function to broadcast 2d namelist variables:

```fortran
subroutine sendnml2()
```
It is possible to write the entire code using the middle and lower layers only. For example, here is the main loop:

500 if (nloop <= itime) go to 2000 ! main interation loop
! add up guard cells in y
   call paguardp(rspace,qe)
isign = -1
! transform charge to fourier space
   if (psolve==PERIODIC_2D) then
     call fft(ftable,rspace,qe,kspace,qt,isign)
   else if (psolve==DIRICHLET_2D) then
     call dblsin(rspace,qe,r2space,q2) ! first double the grid
     call fft(ftable,r2space,q2,kspace,qt,isign)
   endif
! calculate force/charge in fourier space
   call pois(ptable,qt,fxyt,we)
! transform force/charge to real space
   if (psolve==PERIODIC_2D) then
     call fft(ftable,rspace,fxye,kspace,fxyt)
   else if (psolve==DIRICHLET_2D) then
     call fft(ftable,r2space,fxy2,kspace,fxyt)
     call hafdbl(rspace,fxye,r2space,fxy2) ! extract half the grid
   endif
! copy to guard cells
   call pcguardp(rspace,fxye)
call cguardp(nspace,fxye)
   wke = 0.
! push particles
   call push(epart,part,nspace,fxye,bxye,wke)
! move particles into appropriate spatial regions
   call pmove(epart,part,nspace)
! initialize charge density to background
   call sguardp(nspace,qe,qi0)
! deposit charge
   call dpost(epart,part,nspace,qe)
! merge density arrays in x direction
   call aguardp(nspace,qe)
   itime = itime + 1
   go to 500
2000 continue