Frameworks

A framework is a unified environment containing all the components needed for writing code for a specific problem domain. Its goal is the rapid construction of new codes by reuse of trusted modules.

UPIC Framework designed to help construct Plasma PIC calculations by student programmers.

Supports multiple numerical methods, different physics approximations, different numerical optimizations and implementations for different hardware.

Designed to hide the complexity of parallel processing.
Why?

Students typically inherit a code, make changes to only part of it for their thesis.

Often takes a year

• Much must be understood before desired changes can be made safely.

• Many students are beginning programmers

• Rewards are only for new science, not better software

To speed up the process

• Powerful **high level classes** that students can easily use for those parts they do NOT intend to modify

• **Middle layer helper classes** which contain data descriptors for those parts they DO intend to modify

• **Low level utility classes** for debugging, I/O

• Enables student to dig in only as deeply as necessary for those parts they do intend to change.

Example: Cocoa environment (formerly NextStep) on MacOS allows one to include a word processor in an application in less than 50 lines of code.
Layered Approach (software stack)

**Bottom Layer**: optimized Fortran77 kernels.

- Well tested, often with years of use, don’t need to fix it

**Lower Layer**: common utility functions

- Timing, trace facilities, tracking memory leaks
- Parallel I/O

**Middle Layer**: helper objects that work with Fortran90 arrays. They describe data, do not contain data

- Fortran90 arrays powerful, popular, well-understood
- Simpler environment for building new high level objects
- Hides parallel processing details
- Implements some polymorphism
- Can be used in other codes and frameworks

**Higher Layers**: objects with properties

- Objects contain hidden pointers to all structures need to perform required operation
- Send message to object: “FFT yourself.”
Lower Level Classes in UPIC Framework

Errors class

This lower level “class” provides support for debugging

Global variable monitor controls level of debugging:

- monitor = 0, no debug, maximum performance
- monitor = 1, standard debug, check input arguments, memory allocation by class
- monitor = 2, maximum debug, enable trace facility, sanity checks on output

subroutine set_monitor(monval)  ! set debug level

Centralized error handling

All error message are routed into a central routine. This allows one to change error handling, e.g., write to a file, pop up a dialog box, etc.

subroutine set_eunit(iunit)  ! set error unit
subroutine set_edefault(newdefault)  ! error behavior
subroutine ehandler(how,estr)  ! error handler
subroutine werrfl(estr)  ! print error info
Global memory allocation tracking

Keep track of all dynamic memory allocation and deallocation, but not where it occurs. Memory leaks detected but not localized.

```fortran
subroutine clr_alloc() ! clear memory status
subroutine count_alloc(num, atype) ! track memory
subroutine pr_alloc() ! print memory status
```

Typical usage:
```
allocate(mixup(nxhy)); call count_alloc(size(mixup),INTS)
```

Local memory allocation tracking

Keep track of all dynamic memory allocation and deallocation, including where it occurs on a class basis. Uses a link list of memory tables.

```fortran
subroutine count_malloc(name, num, atype) ! track mem
subroutine pr_memtables() ! print memory status
subroutine del_memtables() ! delete memory tables
function get_memtable(name) result(table)
subroutine count_memalloc(this, num, atype) ! track mem
```

Utility functions

```fortran
function get_funit(start) ! get Fortran unit
subroutine wtimer(time, dtime, icntrl) ! obtain time
```
Here are the implementations of a few of the functions. The error handler currently either quits on error or merely logs the error and continues.

subroutine ehandler(how,estr)
! this subroutine handles errors and optionally logs error message
! how = keyword on how to handle error
! estr = optional error message string to be logged in error file
    integer, intent(in) :: how
    character(len=*), intent(in), optional :: estr
    if (present(estr)) write (eunit,*), trim(estr)
    if (how==QUIT) stop
    EXCEPTION = NOERR
end subroutine ehandler

function get_funit(start) result(funit)
! this function returns an unconnected fortran unit number,
! starting with unit = start. returns -1 if none found
    integer, intent(in) :: start
    integer :: funit
! local data
    integer :: i
    logical :: connected
    funit = -1
! check connection status
    do i = start, 99
        inquire(unit=i,opened=connected)
        if (.not.connected) then
            funit = i
            exit
        endif
    enddo
end function get_funit
Usage: **first time** used, obtain current time
call wtimer(time,dtime,-1)
Usage: **subsequently**, update current and returns elapsed time.
call wtimer(time,dtime)

```fortran
subroutine wtimer(time,dtime,icntrl)
! this subroutine performs **wall clock timing**
! time = maximum/minimum elapsed time in seconds
! dtime = current time
! icntrl = (-1,0,1) = (initialize,ignore,read) clock
implicit none
real, intent(out) :: time
integer, intent(inout) :: dtime
integer, intent(in), optional :: icntrl
! local data
integer :: ltime, jclock, count_rate, count_max
character(len=8), save :: sname = ':ptimer:'
ltime = 1
if (present(icntrl)) ltime = icntrl
! do nothing
if (ltime==0) then
  return
! initialize clock
else if (ltime==(-1)) then
  call system_clock(dtime,count_rate,count_max)
! read clock and write time difference from last clock initialization
else if (icntrl==1) then
  jclock = dtime
  call system_clock(dtime,count_rate,count_max)
! check if clock is functioning
  if (count_max==0) then
    time = 0.
  else
! check if clock rolled over
    if (dtime < jclock) jclock = jclock - count_max
! calculate elapsed time
    time = real(dtime - jclock)/real(count_rate)
  endif
endif
end subroutine wtimer
```
Parallel class

This lower level “class” provides basic support for parallel processing. Designed for spectral methods with the number of processors equal to a power of 2. If a non-power of 2 number of processors is used, then only the power of 2 subset will be used for the calculations, but other processors can be used for display or input/output.

Global variable \texttt{kstrt} contains \textbf{processor id} + 1. The variable \texttt{noden} contains the ascii representation of the processor id for printing.

Global variable \texttt{ntasks} contains number of shared processors available for \textbf{multi-tasking} (Pthreads). Currently only supported on Macintosh and Linux.

Initialization of \textbf{parallel processing}

\begin{verbatim}
subroutine init_parallel(idproc,id0,nvp)  ! initial parallel
subroutine end_parallel()                        ! close parallel
\end{verbatim}

\textbf{Partitioning}. Functions for finding optimal 1d and 2d partitions

\begin{verbatim}
subroutine fcomp1(nvp,nx,ny)                  ! 1d
subroutine fcomp2(nvp,nx,ny,nz,nvpy,nvpz)    ! 2d
\end{verbatim}
**Parallel IO.** Functions for collecting distributed data and writing to a file on one node, or vice-versa.

```fortran
subroutine writef(f,iunit,nrec,name,it)
subroutine readf(f,iunit,nrec,name)
```

**Parallel timer.** Finds maximum and minimum time on different processors. Synchronizes with a barrier when starting timer.

```fortran
subroutine pwtimer(time,dtime,icntrl)
```

Optimized **reduction operators** (for powers of 2 only)

```fortran
subroutine plsum(f) ! sum reduction of array
subroutine plmax(f) ! max reduction of array
```
Here is the implementation of one of the functions. Note it is a wrapper function. The low level Fortran77 procedures PPINIT and MPINIT do all the actual work. Note how the trace facility is implemented if monitor=2.

    subroutine init_parallel(idproc,id0,nvp)
    ! this subroutine initializes parallel processing
    ! idproc = processor id in lgrp communicator
    ! id0 = processor id in MPI_COMM_WORLD
    ! nvp = number of real or virtual processors obtained in lgrp
    ! communicator (always a power of 2)
        implicit none
        integer, intent(out) :: idproc, id0, nvp
    ! local data
        character(len=15), save :: sname = ':init_parallel:'
        if (monitor==2) call werrfl(class//sname//' started')
        call PPINIT(idproc,id0,nvp)
    ! kstrt = starting data block number, a global variable
        kstrt = idproc + 1
        write (noden,*) idproc
        noden = trim(adjustl(noden))//':'
    ! ntasks = number of additional tasks, a global variable
        call MP_INIT(ntasks)
        if (ntasks==0) then
            erstr = ' Multi-tasking not supported'
            call ehandler(LOGIT,class//sname//erstr)
        else
            ntasks = ntasks - 1
        endif
        if (monitor==2) call werrfl(class//sname//' complete')
    end subroutine init_parallel
Perrors class

This lower level “class” modifies the error class to provide support for parallel debugging. This class is used throughout the Frameworks. It “inherits” from errors and overrides a few of the functions, adds new ones.

Error handling modifications. If error occurs, gracefully terminates parallel processing. Also adds processor id to error output.

```fortran
subroutine ehandler(how,estr)  ! ends parallel processing
subroutine werrfl(estr)       ! adds processor id
```

Debugging routines for finding **maximum, minimum and sum** of various kinds of arrays across multiple processors. These are used as part of a sanity check when monitor=2

```fortran
subroutine frange(f,label,ns,nx,ms,ny)
```

Debugging routines for finding the **difference** of two arrays across multiple processors. These are used primarily for debugging.

```fortran
subroutine ffdiffe(f,g,label,epsmax,meps,iunit,ns,nx,ms,ny,ierr)
```
Here are the implementations of a few of the functions. Note how the perrors class redefines the ehandler and werrfl functions

```fortran
module perrors_class
use errors_class, sehandler => ehandler, swerrfl => werrfl
use parallel_class

subroutine ehandler(how,estr)
! this subroutine handles errors and optionally logs error message
! how = keyword on how to handle error
! estr = optional error message string to be logged in error file
   integer, intent(in) :: how
   character(len=*) , intent(in), optional :: estr
   if (present(estr)) write (eunit,*), trim(noden)//trim(estr)
   if (how==QUIT) then
     call end_parallel()
     stop
   endif
   EXCEPTION = NOERR
end subroutine ehandler

subroutine werrfl(estr)
! this subroutine prints errors, annotated with processor id
! estr = optional error message string to be logged in error file
   character(len=*) , intent(in) :: estr
   write (eunit,*), trim(noden)//trim(estr)
end subroutine werrfl
```
Spect2d Class

This lower level class encapsulates basic information about simulations which use 2d spectral methods for solutions.

type spect2d
    integer :: indx, indy, nvp, mshare, psolver, inorder
end type spect2d

indx/indy = exponent for length in x/y direction,
nvp = number of real or virtual processors
mshare = shared memory architecture flag
psolver = type of poisson solver
inorder = interpolation order

Defines global constants for specifying various boundary conditions for spectral fields.

Constructors for creating descriptors of 2d spectral data

  subroutine new_spect(this,indx,indy,nvp,mshare,psolve,
                        order)                  ! new descriptor
  subroutine new_spect2d(this,spect,order)   ! make clone

Accessor functions to obtain encapsulated information

  function get_ifbc(this)       ! field boundary condition
  function get_ipbc(this)       ! particle boundary condition
  subroutine printout(this)     ! print class data members
Here is the implementation of one of the functions.

```fortran
function get_ifbc(this) result(ifbc)
! this function returns field boundary flag for given field solver
! this = spect2d descriptor data
! ifbc = field boundary flag (PERIODIC_FBC_2D=1,BOUNDED_1G_2D=2)
  implicit none
  type (spect2d) :: this
  integer :: ifbc

! local data
  character(len=10), save :: sname = ':get_ifbc:'
  if (monitor==2) call werrfl(class//sname//' started')
  if (this%psolver==PERIODIC_2D) then
    ifbc = PERIODIC_FBC_2D
  else if (this%psolver==DIRICHLET_2D) then
    ifbc = BOUNDED_1G_2D
  else
    ifbc = 0
    erstr = ' invalid psolver'
    SPECT2D_ERR = 1; EXCEPTION = EXCEPTION + 1
    call ehandler(EDEFAULT,class//sname//erstr); return
  endif
  if (monitor==2) then
    write (erstr,*) 'ifbc = ', ifbc
    call werrfl(erstr)
    call werrfl(class//sname//' complete')
  endif
end function get_ifbc
```