The Center for Astrophysical Thermonuclear Flashes

Application II

Chris Daley

23rd June

An Advanced Simulation & Computing (ASC)
Academic Strategic Alliances Program (ASAP) Center
at The University of Chicago

Creating New Problems

- A new FLASH problem is created by making a directory for it in FLASH3/source/Simulation/SimulationMain. This is where the setup script looks for the problem specific files.
- The source files in a simulation directory that a user will need to modify are:
  - Simulation_data.F90: Fortran module which stores data and parameters specific to the Simulation.
  - Simulation_init.F90: Fortran routine which reads the runtime parameters, and performs other necessary initializations.
  - Simulation_initBlock.F90: Fortran routine for setting initial conditions in a single block.
  - Simulation_initSpecies.F90: Optional Fortran routine for initializing species properties if multiple species are being used.
- Custom implementation of any kernel routine in FLASH can be placed here.

Simulation_data

- A Fortran module containing all data specific to the simulation unit.
- All names should be prefixed with sim_ to make it clear that data belongs to the simulation unit.
- Remember to use the save attribute to prevent data going out of scope.

```
module Simulation_data
  implicit none
  real, save :: sim_pAmbient, sim_xAngle, sim_yAngle, sim_zAngle
end module Simulation_data
```

Simulation_init

- Initializes the simulation unit.
  - Called once at the beginning of the simulation in both new and restarted application runs.
  - Eliminates the need for FLASH2 "*if (firstcall)*" code fragments.

- Example usage:
  - Stores runtime parameter values in Simulation_data private variables.
  - Calculates any runtime parameter derived quantities.
  - Reads a lookup table from a file.

The Config file and Simulation_init

Config file declares the runtime parameters.

```
D sim_pAmbient      Initial ambient pressure
PARAMETER sim_pAmbient     REAL    1.E-5
```

Simulation_init extracts the value of runtime parameters.

The runtime parameter's default value can be overridden in a flash.par

```
subroutine Simulation_init(myPE)
  use Simulation_data
  use RuntimeParameters_interface, ONLY : &
      RuntimeParameters_get

  implicit none
#include "constants.h"
#include "Flash.h"

  integer, intent(in) :: myPE
  call RuntimeParameters_get('sim_pAmbient', &
                             sim_pAmbient)
end subroutine Simulation_init
```

Simulation_initBlock

- Applies initial conditions to the physical domain
  - Initializes Grid data one block at a time.
  - Only called in new application runs (not in restarts).
- Block abstraction allows it to be used with different Grid implementations
  - Called once in UG simulations.
  - Called many times in AMR simulations.
- Generating an initial grid in AMR simulations:
  - Simulation_initBlock is applied to all blocks at the base refinement level.
  - Grid unit refines blocks if refinement criteria met.

Repeats {

    - Simulation_initBlock is re-applied to all blocks.

Simulation_initBlock: Finding cell types

- The Grid API contains a portable way to find the internal cells and guard cells in a particular block.
  - Essential for NFBS Uniform grid mode where block sizes are not always the same size.

Grid_getBlkIndexLimits(blockId, blkLimits, blkLimitsGC, optional: gridDataStruct)

- The arrays *blkLimits* and *blkLimitsGC* contain the lower and upper bounds of a block. For cell-centered PARAMESH data:

blkLimits(LOW,IAXIS)=NGUARD+1; blkLimits(HIGH,IAXIS)=NXB+NGUARD
blkLimitsGC(LOW,IAXIS)=1; blkLimitsGC(HIGH,IAXIS)=NXB+2*NGUARD

- The input argument *gridDataStruct* specifies the underlying grid datastructure, e.g. cell-centered, face-centered, scratch data structure.

Simulation_initBlock: Accessing each cell

- Many Grid API functions available to read / write Grid data:
  - Grid_getPointData, Grid_putPointData
  - Grid_getRowData, Grid_putRowData
  - Most general is Grid_getBlkPtr:

Grid_getBlkPtr(blockID, dataPtr, optional: gridDataStruct)

- Sets the pointer *dataPtr* to the block indicated by *blockID* for the data structure *gridDataStruct*. Free the pointer using Grid_releaseBlkPtr (has same arguments as Grid_getBlkPtr).

- To obtain actual cells coordinates use Grid_getCellCoords:
  Grid_getCellCoords(axis, blockID, edge, guardcell, coordinates, size)

- This stores coordinates for the cells on axis *axis* (IAXIS, JAXIS, KAXIS) at cell location *edge* (LEFT_EDGE, RIGHT_EDGE, CENTER) in the array *coordinates(size)*.

Excerpt from a Simulation_initBlock

```
subroutine Simulation_initBlock(blockID, myPE)
...
call Grid_getBlkIndexLimits(blockID,blkLimits,blkLimitsGC)
sizeX = blkLimitsGC(HIGH,IAXIS) - blkLimitsGC(LOW,IAXIS) + 1 !Num cells inc. guard.
allocate(xCoord(sizeX))
call Grid_getCellCoords(IAXIS, blockID, CENTER, .true., xCoord, sizeX)

call Grid_getBlkPtr(blockId,solnData)
!Loop over each internal cell and initialize data
...
do  i = blkLimits(LOW,IAXIS), blkLimits(HIGH,IAXIS)
   If (xCoord(i) > sim_xpos) solnData(DENS_VAR,i,j,k) = …
end do
call Grid_releaseBlkPtr(blockID,solnData)

end subroutine Simulation_initBlock
```

Simulation_initSpecies

- Implementation only required when working with multiple species.
  - Called from Multispecies_init to initialize fluid properties.
  - Called in new and restarted application runs.
  - Called before Simulation_init.

- General purpose Simulation_initSpecies implementations are available for nuclear networks and ionization (See Simulation /SimulationComposition directory).

- May want to create derived quantities in Simulation_init from the fluids initialized in Simulation_initSpecies.

The Config file and Simulation_initSpecies

Config file declares the species.

```
SPECIES FLD1
SPECIES FLD2
```

Simulation_initSpecies initializes fluid properties.

```
subroutine Simulation_initSpecies()
  use Multispecies_interface, ONLY : Multispecies_setProperty

  implicit none
#include "Flash.h"
#include "Multispecies.h"

  call Multispecies_setProperty(FLD1_SPEC, A, 1.)
  call Multispecies_setProperty(FLD1_SPEC, Z, 1.)
  call Multispecies_setProperty(FLD1_SPEC, GAMMA, &
 1.66666666667e0)

  call Multispecies_setProperty(FLD2_SPEC, A, 4.0)
  call Multispecies_setProperty(FLD2_SPEC, Z, 2.0)
  call Multispecies_setProperty(FLD2_SPEC, GAMMA, 2.0)

end subroutine Simulation_initSpecies
```

Working with block lists

- A single processor contains some portion of the total grid data in one or more blocks.
  - Possible to access data in a grid-package specific way.
  - However, we recommend using Grid API functions so that code is independent of a particular grid-package.

Grid_getListOfBlocks(blockType, listofBlocks, count, optional: refinementLevel)

- Returns the actual block IDs in *listOfBlocks* and the number of block IDs in *count.* The returned block IDs must satisfy the criteria set by *blockType* and *refinementLevel* input arguments.

- NOTE: Any code using this function must “use” the function prototype because this function has an optional argument.

Particle initialization

- Pre-defined particle initialization available:
  - Regular lattice based distribution.
  - Density based distribution - more particles where the density is higher.
- May want to define your own particle initialization.
  - Create a pt_initPositions.F90 in your simulation directory.
- Key variables for particle initialization:
  - *pt_maxPerProc*: Maximum number of particles that can exist on a single processor.
  - *pt_numLocal*: Number of particles currently initialized on this processor.
- A valid initialization requires: *pt_numLocal* <= *pt_maxPerProc*.

Particle initialization

- Normally particle initialization happens after we have laid down our initial grid.
  - A significant clustering of particles can make it hard to satisfy *pt_numLocal* < *pt_maxPerProc*!
- But we can influence the refinement pattern of the initial AMR grid by refining on particle count.
  - Set *refine_on_particles_count* = *.true.* and *max_particles_per_blk* = *value* in flash.par.
  - FLASH will abort if *max_particles_per_blk* criterion not satisfied when we reach *lrefine_max*.
  - Can be used on its own or in conjunction with the standard refinement criteria in Grid_markRefineDerefine.

Code pattern for particle initialization

```
subroutine pt_initPositions(blockID, success)
    ...
    do i = 1, sim_globalNumParticles
       particlePosition = ... !Generate a position for particle i, i.e. from a file or function.

  !work out if particlePosition is within the bounding box of blockID.
       if (isInBlock) then
          if (pt_numLocal + 1 > pt_maxPerProc) then
             success = .false. ; return !Exceeded max # of particles/processor.
          end if
          pt_numLocal = pt_numLocal + 1 !Retains value between pt_initPositions calls.
          particles(BLK_PART_PROP,pt_numLocal) = real(blockID)
          particles(...,pt_numLocal) = … !Some initialization of particle array fields.
       end if
    end do
    success = .true. !Successful initialization of particles on this block.
end subroutine pt_initPositions
```