FLASH Infrastructure Code Units:
Driver, Grid, IO

Flash Tutorial at RAL
May 30, 2012
Klaus Weide

The University of Chicago
Infrastructure Code Topics

- Driver Unit
  - Overview and Function
  - Unsplit vs Split

- Grid Unit
  - Overview: Implementations
  - Overview: blocks, cells,
  - PARAMESH: oct-tree
  - Data structures and Meta-Data
  - Configuring Variables for Grid Data Structures
  - Dimensions and Geometries
  - What the Grid Code Unit Does
  - Filling Guard Cells and Boundary Conditions

- IO Unit preview
Driver Unit

- Overview and Function
- Unsplit vs Split
Driver - Overview and Function

All other code units and their subroutines are called, directly or indirectly, from Driver. There are three phases encompassing everything FLASH does:

**Initialize – Simulate** (producing some output, …) – **Finish**

The main F90 program, Flash.F90, invokes the rest of the code like this:

- call Driver_initFlash
  - Initialize parameters, data, Grid incl. variable values, …
- call Driver_evolveFlash
  - Advance in time (the only kind of “evolution” that FLASH does)
- call Driver_finalizeFlash
  - Clean up nicely
FLASH4 provides two variants of time evolution (two Driver “implementations”): Split and Unsplit.

- Pick the right one for the Hydro implementation used
  (normally this is automatically done by the ./setup command)
- Driver_evolveFlash implements the main loop of FLASH.
- The loop ends normally when one of several conditions is satisfied:
  - Loop counter dr_nstep = nstart ... nend
  - Simulation time reaches tmax
  - Wall clock reaches wall_clock_time_limit
- Time step dt can vary between dtmin and dtmax,
  Driver_computeDt computes new dt after each loop iteration.
- Driver_computeDt calls Hydro_computeDt, Particles_computeDt, etc. to honor time step requirements of different code units.
Time Evolution - Unsplit vs Split

- DriverMain/Split/ Driver_evolveFlash loop for
  split Hydro (PPM, default)
  
  Do ...
    call Hydro(…,SWEEP_XYZ)
    call other physics
  ....
    call Hydro(…,SWEEP_ZYX)
    call other physics
  ....
  End Do

- Each loop iteration advances the solution by 2 dt

- DriverMain/Unsplit/ Driver_evolveFlash loop for
  unsplit Hydro (staggered mesh
  MHD, etc.)
  
  Do ...
    call Hydro(…)
    call other physics
  ....
  End Do

- Each loop iteration advances the solution by dt
Grid Unit

- Overview: Purpose
- Overview: Implementations
- Overview: blocks, cells, ...
- PARAMESH: oct-tree
- Data structures and Meta-Data
- Configuring Variables for Grid Data Structures
- Dimensions and Geometries
- What the Grid Code Unit Actually Does
- Filling Guard Cells
- Boundary Conditions
First Look at Paramesh (and UG) Grids

- Purpose of the Grid: represent data
  - Much more on UNK variables etc. below
  - More precisely, will be talking about the GridMain subunit of Grid

- Each block of data resides on exactly one processor* (at a given point in time)

- At a given point in time, the number of local blocks on a processor lies between 1 and MAXBLOCKS. (or can be 0, at least during initialization)
  - MAXBLOCKS is defined at setup time. This represents a hardwired limit on how many blocks can exist in total.
  - Grid_getLocalNumBlks() returns the current local value.
  - Paramesh attempts to balance blocks across processors so that processor will have approximately equal amounts of work to do.
  - With the FLASH4 Uniform Grid (UG), the number of blocks is always one per processor.

*On notation: processor here means, more correctly: MPI task.
Overview: Implementations

- **UG – Uniform Grid**
  - Fast, very little overhead
  - Use when your problem does not profit from varying resolution

- **Paramesh2 – old AMR for FLASH2 compatibility**

- **Paramesh4.0** – currently the default Grid Implementation

- **Paramesh4dev**
  - May become the default soon; recommended for large runs.
  - Same functions as PM4.0, users should see no differences in results.
    (known exception: very small differences are possible with face variables.)
  - Performance can differ from PM4.0:
    - Faster in handling grid refinement changes
    - Other Grid operations may be slightly slower

- **Chombo – patch-based library, experimental**

Select implementation: `setup shortcuts +ug, +pm40, +pm4dev`
Reminder: blocks and cells

- The grid is composed of blocks
- FLASH4: In current practice, all blocks are of same size.
- May cover different fractions of the physical domain, depending on a block's resolution.
- Data storage area for each block reserves space for some layers of guard cells.
PARAMESH: An Oct-tree of Blocks

- Paramesh specific design:
  - Block Structured
  - All blocks have same dimensions
  - Blocks at different refinement levels have different grid spacings and thus cover different fractions of the physical domain
  - Fixed sized blocks specified at compile time

- Global block numbers are based on Morton order, approximates “space-filling” behavior. (example numbers for PM2; PM4 is very similar.)

- Storage order within each processor follows this ordering. Re-distribution of blocks after refinement changes, for load balancing.

- Oct-tree in 3D: A node has either 8 children or none. (Quad-tree in 2D, binary in 1D)

- Blocks are of type LEAF, PARENT, or ANCESTOR.

- Data for PARENT and ANCESTOR blocks occupies storage space! (not much in 3D)

In choosing Paramesh, the original FLASH code architects chose simplicity of the Paramesh structure over a patch based mesh.

Flash Center for Computational Science
The University of Chicago
Limits of Paramesh

- PARAMESH is based on blocks, not general patches.

- Limitations imposed by Paramesh:
  - Same number of cells in all blocks
  - Same number of guard cell layers in all blocks, all directions
  - Resolution ("Delta") of a block changes by multiples of 2.
  - Resolution of neighbors differs at most by factor of 2.

(In other words: the local refinement level may change by at most ±1)
How Blocks are Addressed

- At a given time, a block is **globally** uniquely identified by a pair \((\text{proc}, \text{BlockID})\), where
  - \(0 < \text{proc} < \text{numprocs}\)
  - \(1 < \text{BlockID} \leq \text{MAXBLOCKS}\)
- **Locally**, \(\text{BlockID}\) is sufficient to specify a block
  - User code can't directly access remote blocks anyway
- Morton Numbers provide another way to identify blocks **globally**.
  - *(private data of the Grid unit, not exposed to other code at runtime)*
- The global block number of a block determines the index of the block's data in output files (checkpoint, plot files). It is not available to user code during run time.
How Blocks are Stored

- Solution data,
- per-block metadata,
- tree information (for local blocks!)

are stored in F90 arrays declared like this:

```fortran
real, dimension(:,:,:,:,MAXBLOCKS) :: UNK
real, dimension(:,MAXBLOCKS)     :: bnd_box
integer, dimension(:,MAXBLOCKS) :: parent
```

etc. etc.

- MAXBLOCKS is a hardwired constant (from setup time)
- "Inactive" (non-leaf) blocks also use storage
- These structures are internal to the Grid unit and should not be accessed directly by other code.
- Use the appropriate Grid_something() subroutine calls instead! (in particular: Grid_getBlkPtr, Grid_getBlkData)
Grid Data Structures

- **CENTER**: (keywords VARIABLE, SPECIES, MASS_SCALAR)
  - The “normal” way to keep fluid variables: logically cell-centered
  - Kept internally in an array UNK of dimensions UNK(NUNK_VARS,NXB +gcs,NYB+gcs,NZB+gcs,sMAXBLOCKS)

- **FACEX, FACEY, FACEZ**
  - Face-centered variables, currently used by unsplit MHD solver
  - Supported in UG, PM 4.0, PM 4dev

- **SCRATCH**: *(data that is never updated automatically by Grid)*
  - Additional block-oriented storage provided by FLASH (not PM Kernel)
  - Guard cell filling or other communications not supported

- **WORK**: *(only 1 “variable”, not recommended for portability)*
  - Additional block-oriented storage provided by PARAMESH (not in UG)
  - Used internally by some units (currently: geometric multigrid solvers)

- **(FLUX – not a permanent data store, for flux corrections by Hydro)**
Configuring Variables for Grid Data Structures

- Use VARIABLE vvvv in Config for unk(VVV_VAR,:,,:,:,:)**
  - gridDataStruct=CENTER*
- Use SPECIES ssss in Config for unk(SSSS_SPEC,:,,:,:,:)
  - gridDataStruct=CENTER
- Use MASS_SCALAR mmmm for unk(MMMM_MSCALAR,:,,:,:,:)
  - gridDataStruct=CENTER
- Use FACEVAR ffff in Config for facevarx(FFFF_FACE_VAR,:,,:,:,:), facevary(FFFF_FACE_VAR,...), & facevarz(FFFF_FACE_VAR,...)
  - gridDataStruct=FACEX/FACEY/FACEZ (or for some calls: FACES)
- Use GRIDVAR ggg for scratch(:,:,:GGG_SCRATCH_GRID_VAR,:)
  - gridDataStruct=SCRATCH

* Many Grid interfaces have a gridDataStruct argument to specify what kind of data to act on. Examples: Grid_getBlkPointer, Grid_putBlkData, Grid_getBlkIndexLimits, Grid_fillGuardCells. See API documentation of these interface for details.

** The internal organization (order of array indices) is important for code working with block pointers as returned by Grid_getBlkPointer.
Configuring Variables for Grid Data Structures II

- Use VARIABLE vvvv in Config for unk(VVV_VAR,:,:,:,:)
  - gridDataStruct=CENTER
- Use SPECIES ssss in Config for unk(SSSS_SPEC,:,:,:,:)
  - gridDataStruct=CENTER
- Use MASS_SCALAR mmmm for unk(MMMM_MSCALAR,:,:,:,:)
  - gridDataStruct=CENTER

Cell-centered variables from VARIABLE, SPECIES, MASS_SCALAR become parts of the same large array:

- unk(1:NPROP_VARS,:,:,:,:) holds NPROP_VARS VARIABLES
- unk(SPECIES_BEGIN:SPECIES_END,:,:,:,:) holds NSPECIES SPECIES
  - Note: often NSPECIES=0, in that case SPECIES_END=SPECIES_BEGIN-1
- unk(MASS_SCALARS_BEGIN:NUNK_VARS,:,:,:,:) holds NMASS_SCALARS MASS_SCALARs
  - Often NMASS_SCALARS=0, in that case MASS_SCALARS_BEGIN = NUNK_VARS+1
More On Variables for Grid Data Structures

- The “VARIABLE” part of unk represents most solution variables
  - VARIABLE dens TYPE: PER_VOLUME – conserved variable per volume-unit
  - VARIABLE ener TYPE: PER_MASS – energy in mass-specific form
  - VARIABLE temp TYPE: GENERIC – not a conserved entity in any form
  Specify the TYPE correctly to ensure correct treatment in Grid interpolation.
  See Config files in existing code Units for examples: Hydro, Eos, ...

- The SPECIES part of unk represents mass fractions
  - Get automatically advected by Hydro
  - Should probably be used with Multispecies Unit and Multigamma EOS
  - Should always add up to 1.0, code may enforce this
  - Treated as a per-mass variable for purposes of interpolation

- The MASS_SCALAR part of unk represents additional variables
  - Get automatically advected by Hydro
  - Treated as a per-mass variable for purposes of interpolation
Dimensions and Geometries

Geometry Support

The FLASH4 Grid supports these geometries:

- Cartesian - 1D, 2D, 3D
- Cylindrical - 2D, (3D)
- Spherical - 1D, (2D), (3D)
- Polar - (2D)

Combinations in **bold** have been extensively used & tested at the FLASH Center.

*(Note: for a specific application, geometry support may be limited by available solvers!)*

The Grid Implementation:

- Makes used of Paramesh4 support of geometries
- Centralized support by Grid unit, provides routines for cell volumes, face areas, etc.
- Grid uses geometry-aware conservative interpolation at refinement boundaries
  - This is the default interpolation, internally called “monotonic”.
  - We provide a way to use an alternative Grid implementation's native methods instead:
    - `./setup ... -gridinterpolation=native`
- Use setup -3d -geometry= and/or runtime parameter *geometry* in flash.par to specify.
What the Grid Code Unit Actually Does

Note: the following focuses on AMR Grids; UG is simpler.

The Grid unit is responsible for

- Keeping account of the spatial domain as a whole:
  - Extent and size, outer boundaries
- Keeping and maintaining block structure:
  - Which blocks exist?
  - Where are they?
  - Sizes and other properties of blocks
  - Neighbors
  - Parent / child links for AMR
- Initializing block structure:
  - Initialize the metadata and links mentioned above
  - Keep Grid structure valid:
    - Consistent (if A is child of B, then B must be parent of A, etc. etc.)
    - For PARAMESH: no refinement jumps by more than 1 level
What the Grid Unit Actually Does - Cont.

Note: the previous slide was mostly about metadata; now the stuff actually wanted by users...

The *Grid* unit is also responsible for

- Keeping data ("User data", "Solution data", "payload"):  
  - Provide storage  
    - UNK, FACEVAR\{X,Y,Z\}, SCRATCH, (WORK)  
    - FLUXes and other more temporary arrays

- **Initializing** solution data:
  - Actually **left to the user**, who provides a subroutine *Simulation_initBlock()*  
  - *Grid invokes* user function, applies **refinement criteria**, repeat as necessary

- maintaining and keeping track of data during refinement changes:
  - Apply refinement criteria as requested  
  - Copy data within processor, and/or communicate between procs  
  - Involves prolongation (interpolation)  
  - Involves restriction (valid data in PARENT blocks)
What the Grid Unit Actually Does - Cont..

Note: the previous slide was about data and mesh changes; now what's left to do between those changes?

- The Grid unit is **also** responsible for
- Operations that communicate user data between blocks:
  - Prolong (interpolate) data
    - After new leaf blocks are created
  - Restrict (summarize) data
    - PARENT blocks usually get summarized data as part of guard cell filling
  - Flux correction (special operation invoked from Hydro)
  - Edge averaging (special operation invoked from MHD Hydro)

And finally...

- Guard cell filling
  - The most important form of data communication on an established mesh configuration.
  - Called frequently, by various code units
  - May move a lot of data between procs, efficiency is important!
Guard Cell Filling – When

Note: the following is focused on Paramesh4, but the high-level calls apply to all grid implementations

When are guard cells filled?

- **Directly**: High-level call to `Grid_fillGuardCells` (or maybe `amr_guardcell`)
  - Always a global operation involving all processors
  - Usually fills guard cells of LEAF blocks and their parents – but don't count on it for PARENT blocks.

- **Indirectly**: internally as part of some other Grid operation
  - As part of `amr_prolong` (filling new leaf blocks)

- Indirectly during global direct filling:
  - Auxiliary filling of a PARENT block's guard cells in order to provide input for interpolation to this PARENT's child, a finer-resolution LEAF node.
Guard Cell Filling - Usage

When should you fill guard cells?

- Before a subroutine that you have written uses guard cells, you need to make sure they are filled with valid and current data.
- FLASH4 does not guarantee that guard cells are valid on entry to a solver, source term code unit, etc.!

How should you fill guard cells?

- Only worry about direct filling of LEAF guard cells – that is nearly always what is needed.
- Basic high-level call:
  
  Call Grid_fillGuardCells(CENTER_FACES,ALLDIR)

- High-level call with automatic Eos call on guard cells:
  
  Call Grid_fillGuardCells(CENTER_FACES,ALLDIR,doEos=.true.)

  Eos often needs to be called to get cells at refinement boundaries, where data was interpolated, into thermodynamic balance.

- There are many additional optional arguments, see API docs. They are for increasing performance, and can all be initially ignored.
Blocks consist of cells: guard cells and interior cells.

For purposes of guard cell filling, guard cells are organized into guard cell regions.

During guard cell filling, each guard cell region may get filled from a different data source:

- A local neighbor block
- A remote neighbor block
- A boundary condition
  - using data from adjacent interior cells
  - Using fixed or coordinate-based data
- Interpolation from parent (if the block touches a fine/coarse boundary)

In PARAMESH4, diagonal regions are treated just like “face neighbor” regions.
For purposes of guard cell filling, guard cells are organized into guard cell regions.

During guard cell filling, each guard cell region may get filled from a different data source:

- A local neighbor block
- A remote neighbor block
- A boundary condition
  - using data from adjacent interior cells
  - Using fixed or coordinate-based data
- Interpolation from parent (if the block touches a fine/coarse boundary)
Filling guard cells Ia

- For purposes of guard cell filling, guard cells are organized into guard cell regions.
  - In 2D, a block has 8 guard cell regions.
  - In 3D, a block has 26 guard cell regions!

- During guard cell filling, each guard cell region may get filled from a different data source:
  - A local neighbor block
  - A remote neighbor block
  - A boundary condition
    - using data from adjacent interior cells
    - Using fixed or coordinate-based data
  - Interpolation from parent (if the block touches a fine/coarse boundary)
For purposes of guard cell filling, guard cells are organized into guard cell regions.

In 2D, a block has 8 guard cell regions. In 3D, a block has 26 guard cell regions!

During guard cell filling, each guard cell region may get filled from a different data source:
- A local neighbor block
- A remote neighbor block
- A boundary condition
  - using data from adjacent interior cells
  - Using fixed or coordinate-based data
- Interpolation from parent (if the block touches a fine/coarse boundary)

face direction
diagonal direction
For purposes of guard cell filling, guard cells are organized into guard cell regions.

In 2D, a block has 8 guard cell regions.

In 3D, a block has 26 guard cell regions!

During guard cell filling, each guard cell region may get filled from a different data source:

- A local neighbor block
- A remote neighbor block
- A boundary condition
  - using data from adjacent interior cells
  - Using fixed or coordinate-based data
- Interpolation from parent (if the block touches a fine/coarse boundary)

-1,-1
-1,0
-1,1
0,-1
0,0
0,1
1,-1
1,0
1,1

face neighbor
diagonal neighbor
For purposes of guard cell filling, guard cells are organized into guard cell regions.

During guard cell filling, each guard cell region may get filled from a different data source:

- A local neighbor block
- A remote neighbor block
- A boundary condition
  - using data from adjacent interior cells
  - Using fixed or coordinate-based data
  - Interpolation from parent (if the block touches a fine/coarse boundary)
Filling guard cells at Boundary I

- For purposes of guard cell filling, guard cells are organized into guard cell regions.

Now assume a block at the corner of the domain:

- During guard cell filling, each guard cell region may get filled from a different data source:
  - A local neighbor block
  - A remote neighbor block
  - A boundary condition
    - using data from adjacent interior cells
    - Using fixed or coordinate-based data
  - Interpolation from parent (if the block touches a fine/coarse boundary)

Domain boundaries
For purposes of guard cell filling, guard cells are organized into guard cell regions. The guard cell regions in red represent locations outside of the domain.

During guard cell filling, each guard cell region may get filled from a different data source:

- A local neighbor block
- A remote neighbor block
- A boundary condition
  - using data from adjacent interior cells
  - Using fixed or coordinate-based data
  - Interpolation from parent (if the block touches a fine/coarse boundary)
Filling guard cells at Boundary III

- For purposes of guard cell filling, guard cells are organized into guard cell regions.

- During guard cell filling, each guard cell region may get filled from a different data source:
  - A local neighbor block
  - A remote neighbor block
  - A boundary condition
    - using data from adjacent interior cells
    - Using fixed or coordinate-based data

- Grid_bcApplyToRegionSpecialized is called and passed a pointer to the data in the blue region.
  (actually, to a copy of the block data)
For purposes of guard cell filling, guard cells are organized into guard cell regions.

During guard cell filling, each guard cell region may get filled from a different data source:

- A local neighbor block
- A remote neighbor block
- A boundary condition
  - using data from adjacent interior cells
  - Using fixed or coordinate-based data

Grid_bcApplyToRegionSpecialized may fill in the guard cell region.

OR it may decline to handle this, and then:

The subroutine Grid_bcApplyToRegion is called and passed a pointer to the data in the blue region.
Implementing Boundary Conditions

- **Grid_bcApplyToRegionSpecialized** gets called first
  - This is normally a no-op stub
  - This is the preferred place to users to hook in **customized implementations**.
  - May decide to handle the call, based on BC type, direction, ...
  - Before returning, sets “applied” flag to signal that the BC was handled.

- **Grid_bcApplyToRegion** gets called if **Grid_bcApplyToRegionSpecialized** did not handle the case.
  - The standard implementation of **Grid_bcApplyToRegion** in source/Grid/GridBoundaryConditions provides the standard simple BC types: REFLECTING, OUTFLOW, DIODE, ...
  - It is a good place to start if you need to write your own!

- Both interfaces provide information that an implementation, can use to fill guard cells at boundaries, including:
  - A block handle (usually, block ID) identifying the block being filled
  - Location of the data region within the Grid block
BCs – Complications

- Grid_bcApplyToRegion* may be called on a non-LEAF block.
- Grid_bcApplyToRegion* may be called on a block that is not even local!
  - This can happen if a parent block needs to be filled to provide input data for interpolation, and the parent resides on a different PE from the leaf.
  - Simple BC methods don't have to be aware of this.
- But if your method depends on coordinate information, or needs to access the block by its ID, beware!
- See source/Grid/GridBoundaryConditions/README and Users Guide in those cases.
- The data region passed to Grid_bcApplyToRegion* is in transposed form:
  - Reference it like regionData(I,J,k,ivar), where
  - I counts cells in the normal direction (NOT always: x direction!),
  - J,K cont cells in the other directions
  - Ivar counts variables
  - This is convenient for implementing simple BC where location does not matter, but complicates things if you need to know where a cell is within the block.
- Use provided examples!
BCs – Simplifications

If you prefer a simpler interface:

- Handle one data row at a time (vector of data in normal direction)
- Powerful enough to implement hydrostatic boundaries
- REQUIRES Grid/GridBoundaryConditions/OneRow (see source files there!)
- Implements a version of Grid_bcApplyToRegionSpecialized
- Provides functions Grid_applyBCEdge, Grid_applyBCEdgeAllUnkVars
- Too customize, user should provide own implementation of Grid_applyBCEdge.F90 (or Grid_applyBCEdgeAllUnkVars.F90)
The IO unit is responsible for:

- Writing checkpoint files
  - For restarting, ...
- Writing "plot files"
  - For visualization
  - For further analysis
- Writing particle data files
  - For visualization and further processing

- Binary output files are writing in a structured format: HDF5, pnetcdf
- Various tools can process FLASH files, including Visit
Grid and other Infrastructure Code

• Questions?