The Center for Astrophysical Thermonuclear Flashes

Overview & Architecture

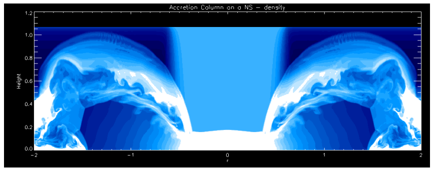
Anshu Dubey

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

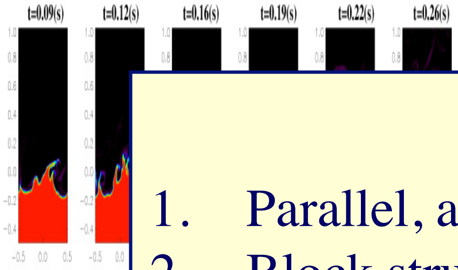
Outline

- FLASH Overview
- Architecture
  - Units
    - UnitMain
    - Subunits
    - Alternate implementations
  - Namespace
  - Inheritance
    - Stubs
  - Setup script
    - Config files

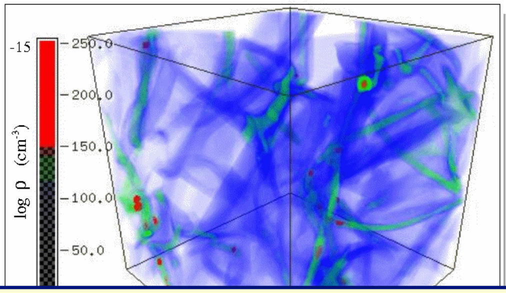
FLASH Capabilities Span a Broad Range…

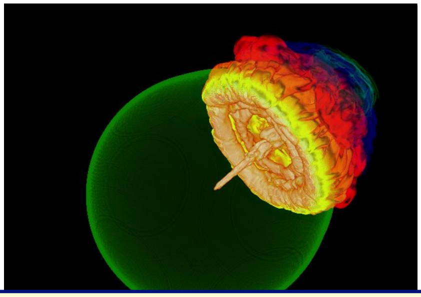


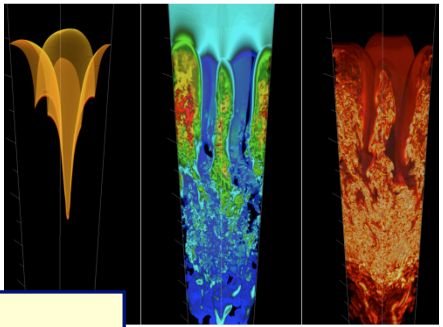
Shortly: Relativistic accretion onto NS



Wave breakin



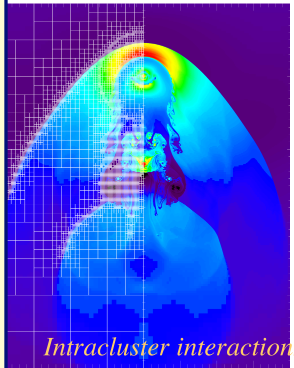




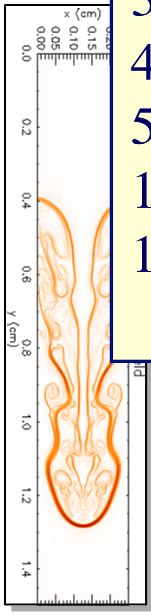
lear Burning

The FLASH code

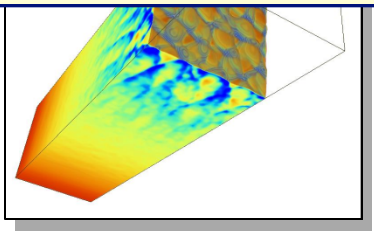
1. Parallel, adaptive-mesh refinement (AMR) code
2. Block structured AMR; a block is the unit of computation
3. Designed for compressible reactive flows
4. Can solve a broad range of (astro)physical problems
5. Portable: runs on many massively-parallel systems
11. Scales and performs well
12. Fully modular and extensible: components can be combined to create many different applications



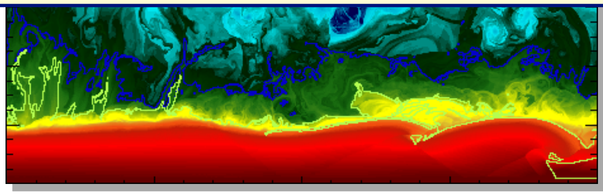
Intracluster interactions



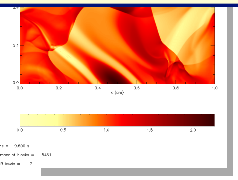
Magnetic Rayleigh-Taylor



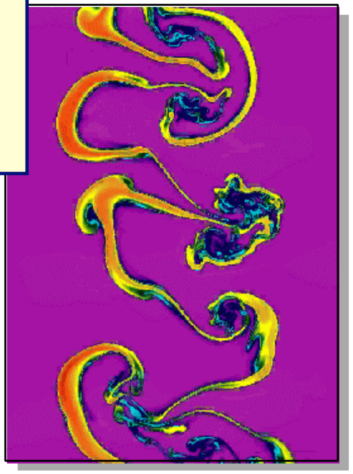
Cellular detonation



Helium burning on neutron stars



Orzag/Tang MHD vortex



Richtmyer-Meshkov instability

FLASH Basics

- An application code, composed of units/modules. Particular modules are set up together to run different physics problems.
- Fortran, C, Python, …
  - More than 500,000* lines of code, 75% code, 25% comments
- Very portable, scales to tens of thousand processors

Capabilities

- Infrastructure
  - Configuration (setup)
  - Mesh Management
  - Parallel I/O
  - Monitoring
    - Performance and progress
  - Verification
    - FlashTest
      - Unit and regression testing

- Physics
  - Hydrodynamics, MHD, RHD
  - Equation of State
  - Nuclear Physics, other Source Terms
  - Gravity
  - Particles, active and passive
  - Material Properties
  - Cosmology
  - New : HEDP, FSI

New Development Directions

- HEDP
  - 3T Model
  - Laser energy deposition
  - Biermann Battery
  - Anisotropic (along B-fields) heat conduction
  - Jacobian-Free Newton-Krylov implicit solver

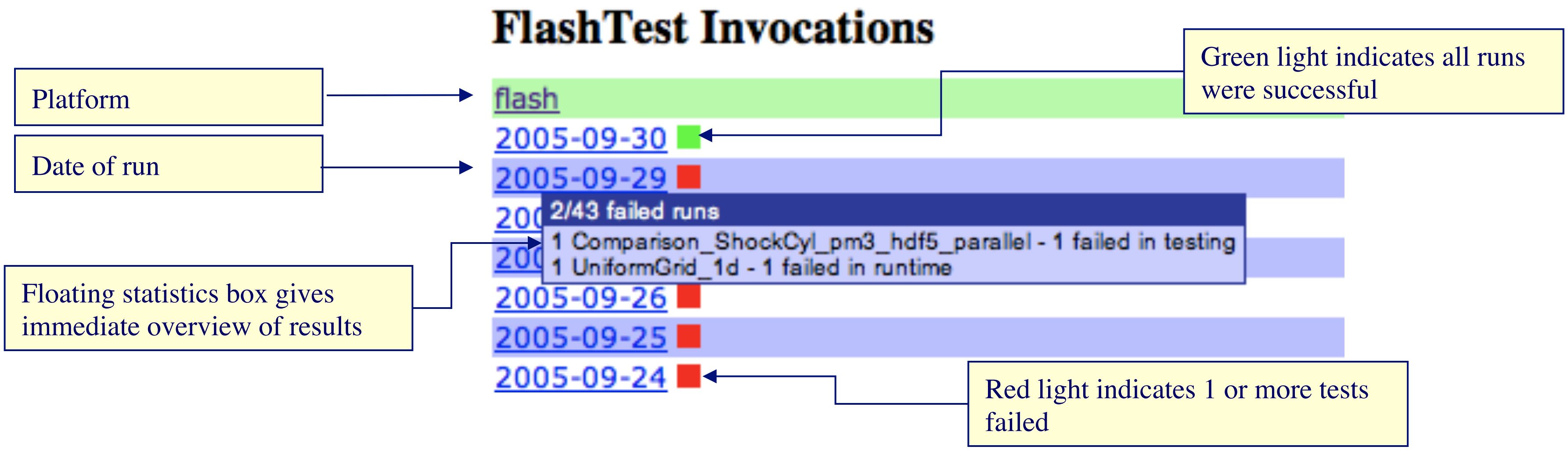
- Fluid-structure Interactions
  - Incompressible Navier-Stokes solver
  - Immersed boundaries
  - Solid bodies modeled with Lagrangian trackers

Auditing Process

- ❑ SVN for Version Control
- ❑ Test Suite
- ❑ Online Coding Violation Tracking and Bugzilla
  - ❑ Unfinished tasks, bugs, bad code, developer queries
- ❑ Profiling Tools
  - ❑ Memory / speed diagnostic tools
  - ❑ External tools like JUMPSHOT / PAPI / TAU
- ❑ Documentation
  - ❑ Online documentation for Unit APIs -- ROBODOC
  - ❑ User’s guide in HTML and PDF
  - ❑ “Howto” available for developers, various platforms
  - ❑ Email users’ group

Verifiability: The Test Suite

- FLASH Test Suite runs a variety of problems to validate the code on a daily basis
- Runs unit tests, and regression tests
- Essential for immediately identifying bugs unintentionally introduced into the code
- Can also track daily code performance



FlashTest Invocations

flash

2005-09-30 🟩

2005-09-29 🟥

200

200

2/43 failed runs

1 Comparison_ShockCyl_pm3_hdf5_parallel - 1 failed in testing

1 UniformGrid_1d - 1 failed in runtime

2005-09-26 🟥

2005-09-25 🟥

2005-09-24 🟥

Platform

Date of run

Green light indicates all runs were successful

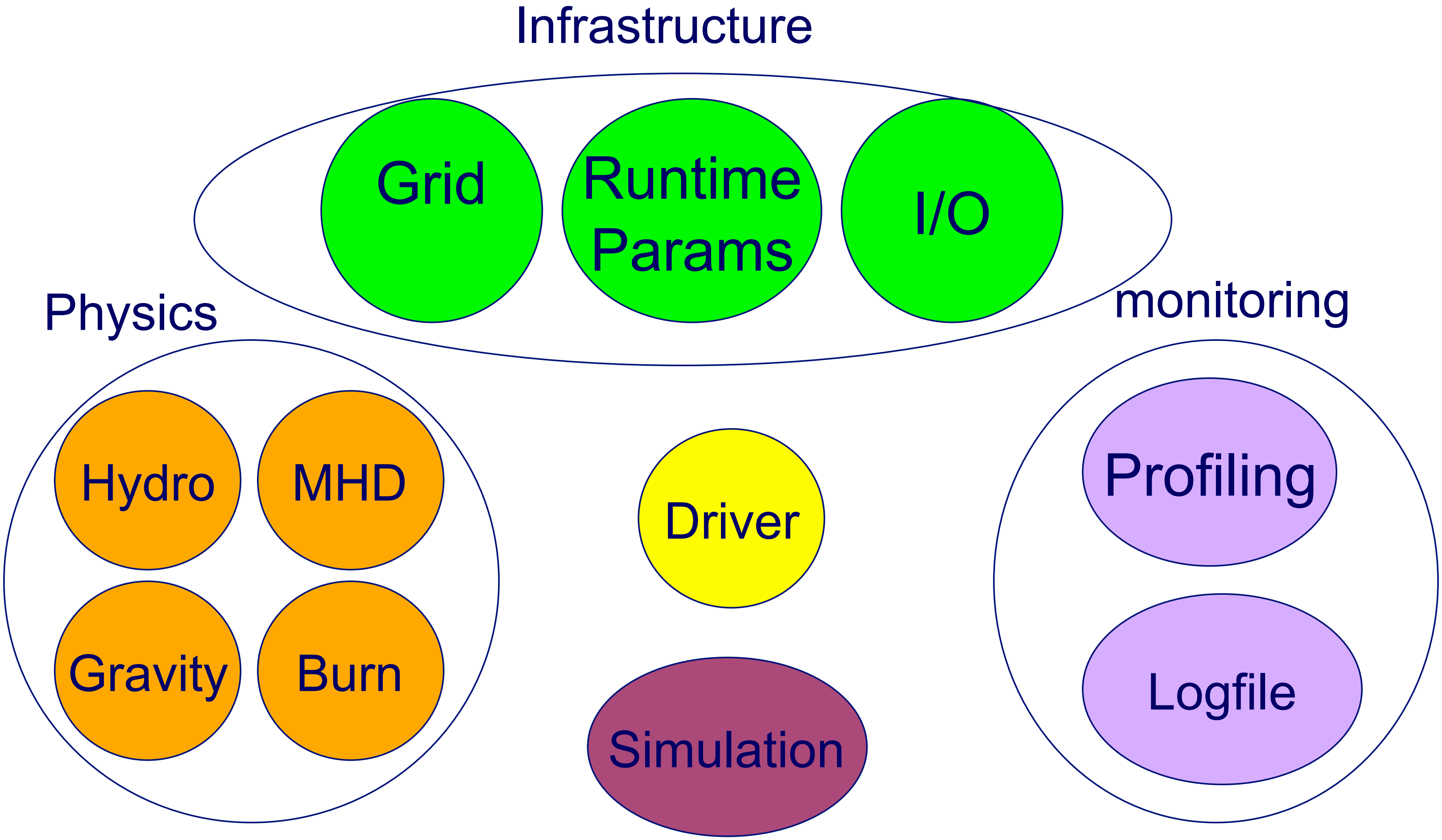
Floating statistics box gives immediate overview of results

Red light indicates 1 or more tests failed

Architecture : Unit

- FLASH basic architecture unit
  - Component of the FLASH code providing a particular functionality
  - Different combinations of units are used for particular problem setups
  - Publishes a public interface (API) for other units' use.
  - Ex: Driver, Grid, Hydro, IO etc
- Fake inheritance by use of directory structure
- Interaction between units governed by the Driver
- Not all units are included in all applications

FLASH Units: Examples



Infrastructure

Grid

Runtime Params

I/O

Physics

Hydro

MHD

Gravity

Burn

Driver

Simulation

monitoring

Profiling

Logfile

Inside a Unit: The Top Level

- First **capitalized** directory in a branch of the source tree is a unit
- Contains **stubs** for every public function (**API**) in the unit
  - Does not contain the data module (unit scope data)
  - Individual API functions may be implemented in different subunits
  - A unit has a minimum three functions in its API, no limit on the maximum
    - Unit_init, Unit_finalize and the “do-er” function for the unit
- If necessary, contains a directory for the **local API**
- May contain the **unit test**
  - Different Unit tests can reside at different levels in the unit hierarchy
- The Config file contains minimal information, no runtime parameters except “useUnit” defined
- **Makefile** includes all the API functions.

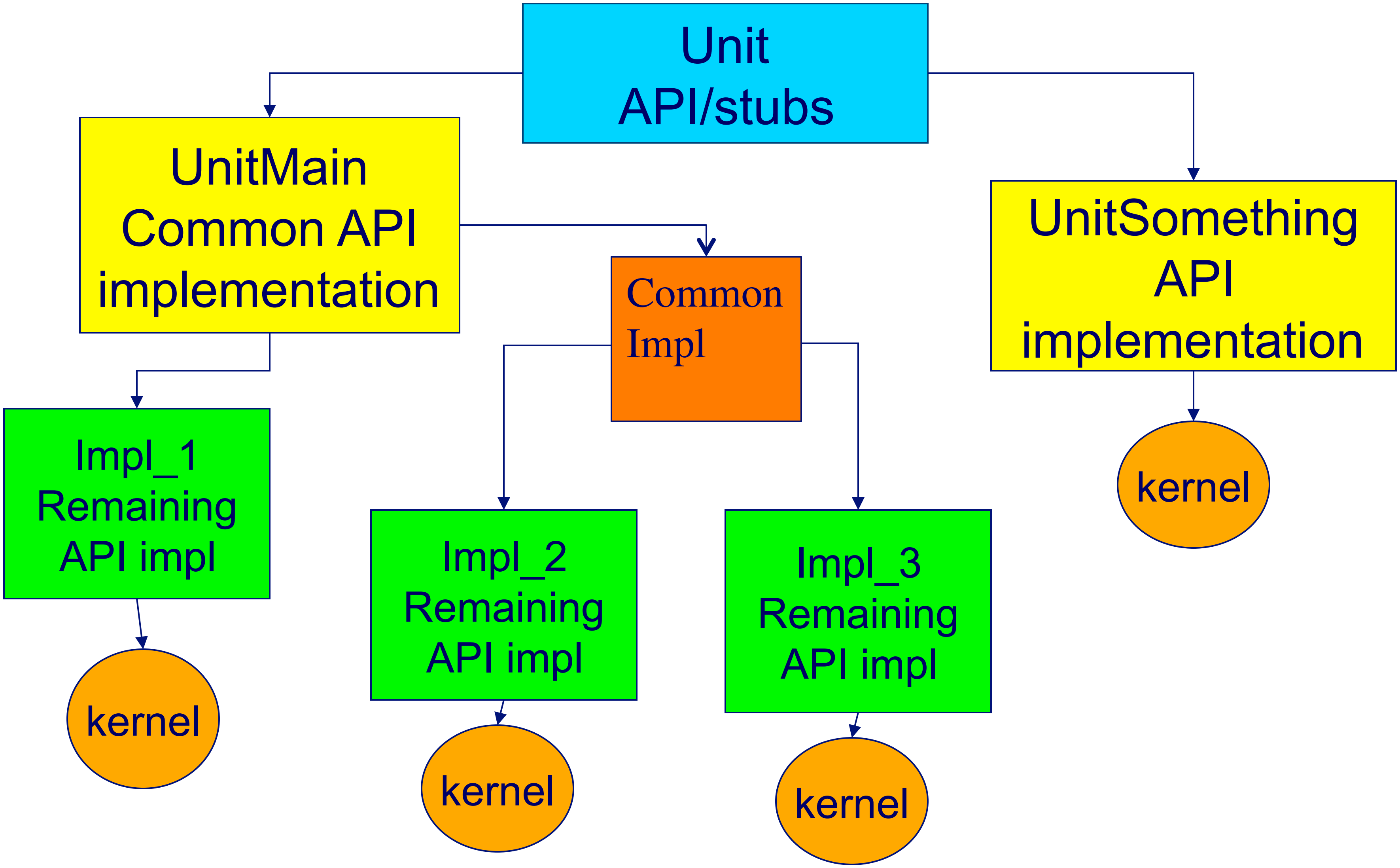
Subunits

- Every unit has a **UnitMain** subunit, which must be included in the simulation if the unit is included.
  - Has implementations for the init, finalize and the main “do-er” function
  - Also contains the unit scope data module
- The API functions and private functions implemented in different subunits are **mutually exclusive**
- Subunits other than UnitMain may have private Unit scope functions that **can be called by other subunits**.
  - un_suInit and un_suFinalize are the most common ones
  - (naming convention explained later)
- Subunits can also have **private data modules**, strictly within the scope limited to the specific subunit
- Subunits can have their own **unit tests**

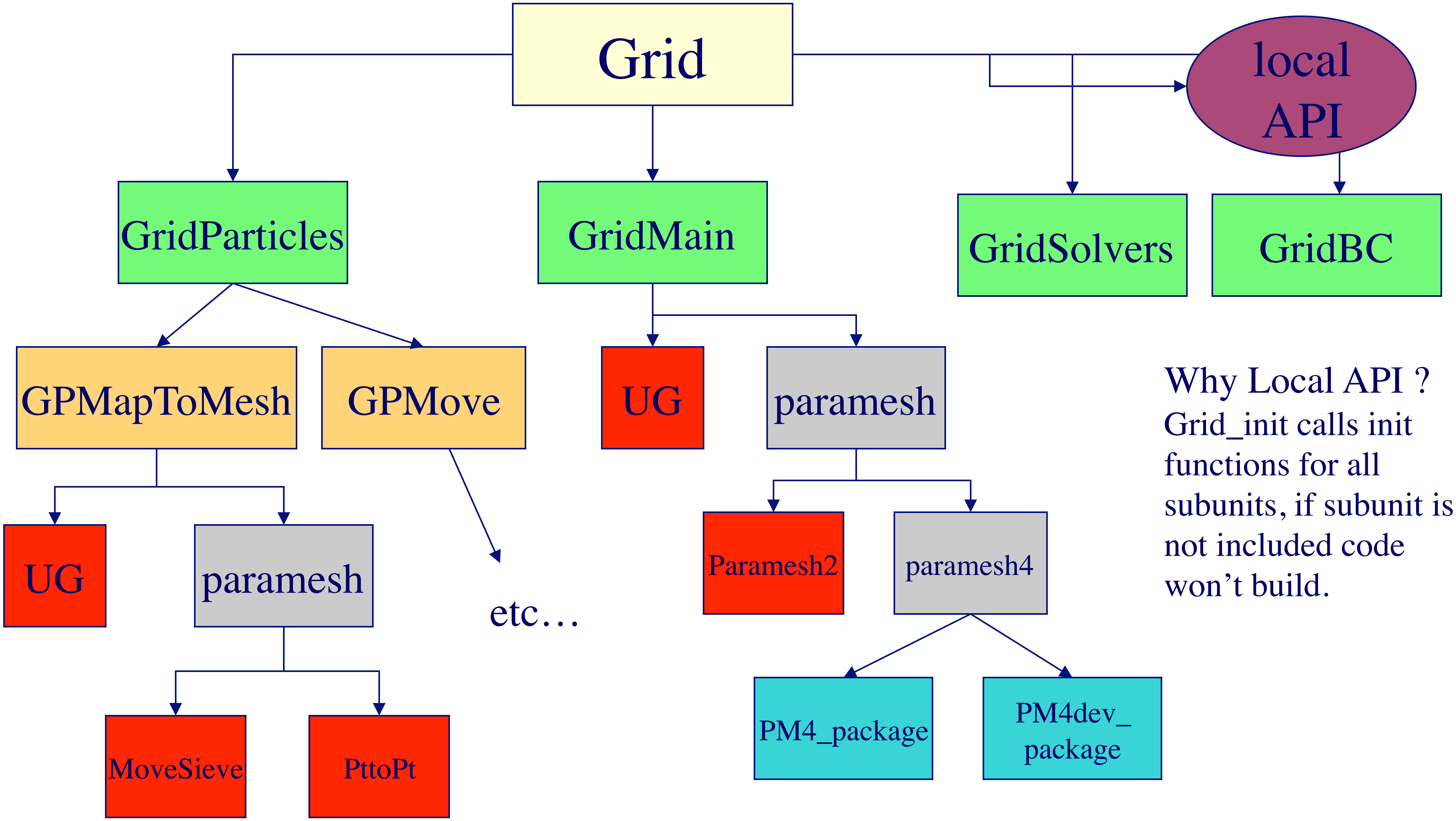
More on Subunits

- A subunit may have multiple alternative implementations
- Alternative implementations of UnitMain also act as alternative implementations of the Unit.
- Some subunits have multiple implementations that could be included in the same simulation
  - GridParticles is one possible example.
  - Alternative implementations are specified using the “EXCLUSIVE” directive
- The “KERNEL” keyword indicates that subdirectories below that level need not follow FLASH architecture, and the entire subtree will be included in the simulation

Unit Hierarchy



Example of a Unit – Grid (simplified)

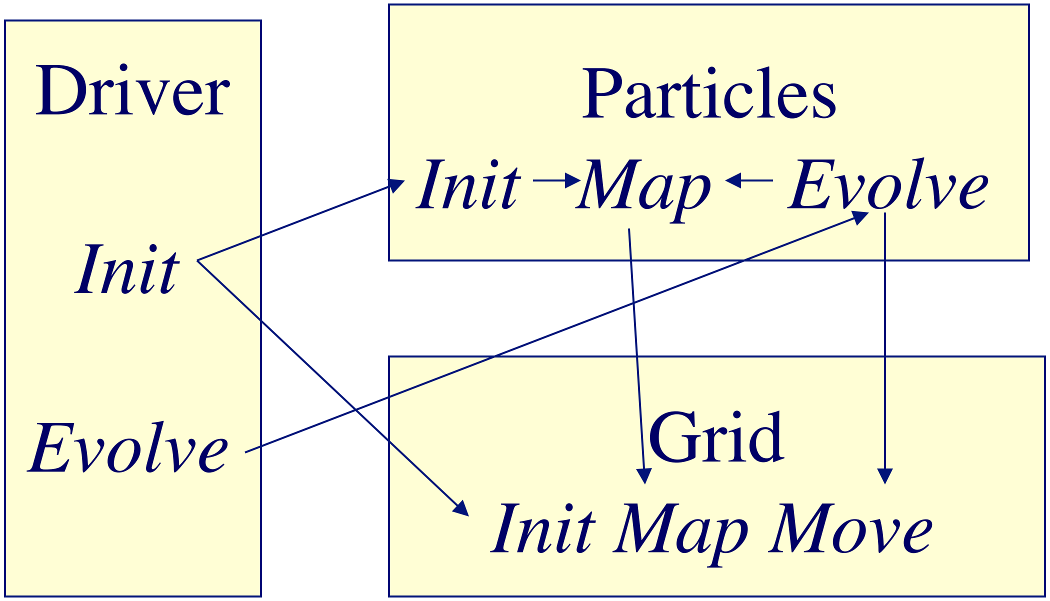


Why Local API ?

Grid_init calls init functions for all subunits, if subunit is not included code won’t build.

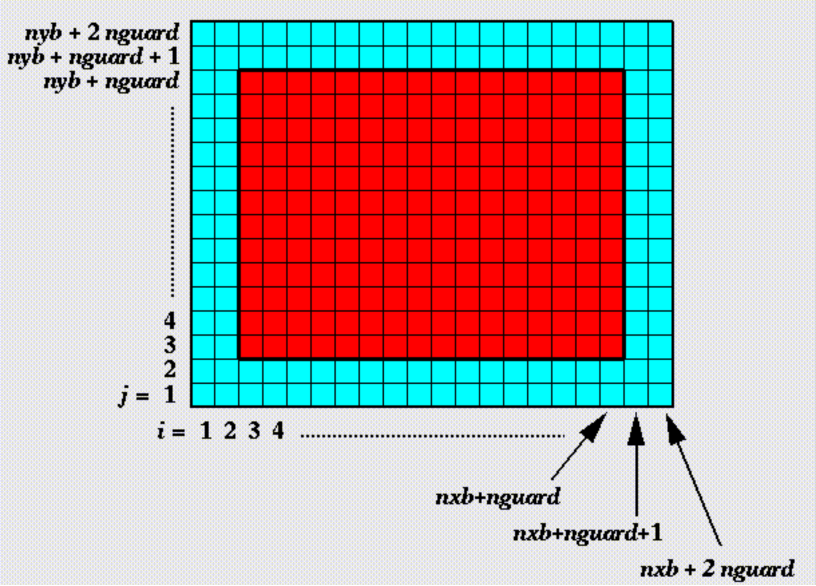
Functional Component in Multiple Units

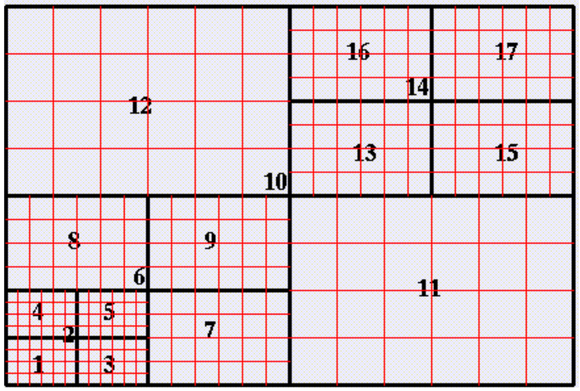
- Example Particles
  - Position initialization and time integration in Particles unit
  - Data movement in Grid unit
  - Mapping divided between Grid and Particles
- Solve the problem by moving control back and forth between units



Basic Computational Unit, Block

- The grid is composed of blocks
- Cover different fraction of the physical domain.
- In AMR blocks at different levels of refinement have different grid spacing.



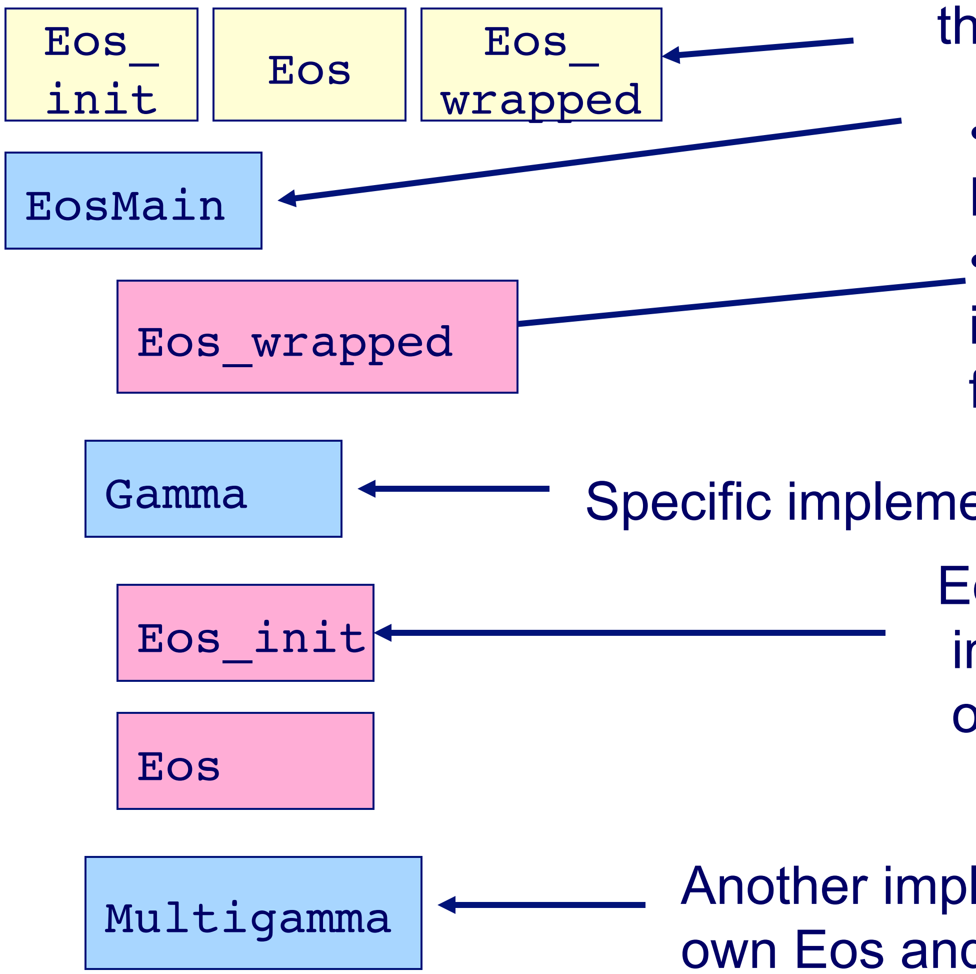


Architecture : Inheritance

- Inheritance implemented through directory structure and Config file directives understood by the setup script
- A child directory inherits all functions from the parent directory
  - If the child directory has its own implementation of a function, it replaces the inherited one.
  - The implementation in the lowest level offspring replaces all implementations in higher level directories.
  - An implementation in the “Simulation/MyProblem” directory overrides all implementations when running MyProblem
- Config files arbitrate on multiple implementations through “Default” keyword
- Runtime environment is created by taking a union of all variables, fluxes, and runtime parameters in Config files of included directories.
  - Value given to a runtime parameter in the “Simulation/MyProblem/Config” overrides any value given to it in other Config files
  - Value in “flash.par” overrides any value given in any Config file

Multiple Config file initial values of a runtime parameter in units other than the simulation unit can lead to non-deterministic behavior since there are no other precedence rules.

Inheritance Through Directories: Eos



•Stub Implementations of the three functions at the top level

- There is only one subunit: Eos/ EosMain
- Replaces the stub with an implementation common to all formulations of EOS

Specific implementation

Eos/EosMain/Gamma implements gamma versions of Eos_init and Eos

Another implementation, which will have its own Eos and Eos_init etc.

Namespace

- Namespace directories are capitalized, organizational directories are not
- All API functions of unit start with Unit_ (i.e.Grid_getBlkPtr, Driver_initFlash etc)
- Subunits have composite names that include unit name followed by a capitalized word describing the subunit (i.e. ParticlesMain, ParticlesMapping, GridParticles etc)
- Private unit functions and unit scope variables are named un_routineName (i.e. gr_createDomain, pt_numLocal etc)
- Private functions in subunits other than UnitMain are encouraged to have names like un_suRoutineName, as are the variables in subunit scope data module

Naming Conventions: Within files

- Constants are all uppercase, usually have preprocessor definition, multiple words are separated by an underscore.
  - Permanent constants in “constants.h” or “Unit.h”
    - #define MASTER_PE 0
    - #define CYLINDRICAL 3
  - Generated by setup script in “Flash.h”
    - #define DENS_VAR 1
    - #define NFACE_VARS 6
- Style within routines
  - Variables from Unit_data start with unit_variable: “eos_eintSwitch”
  - Variables begin lowercase, additional words begin with uppercase: “massFraction”

Naming Conventions – How they help

- The significance of capitalizing unit names:
  - A new unit can be added without the need to modify the setup script.
  - If the setup script encounters a top level capitalized directory without an API function to initialize the unit, it issues a warning.
- Variable Style:
  - Immediately clear if variable is CONSTANT, local (massFraction) or global (eos_eintSwitch) in scope

Setup Script Implements Architecture

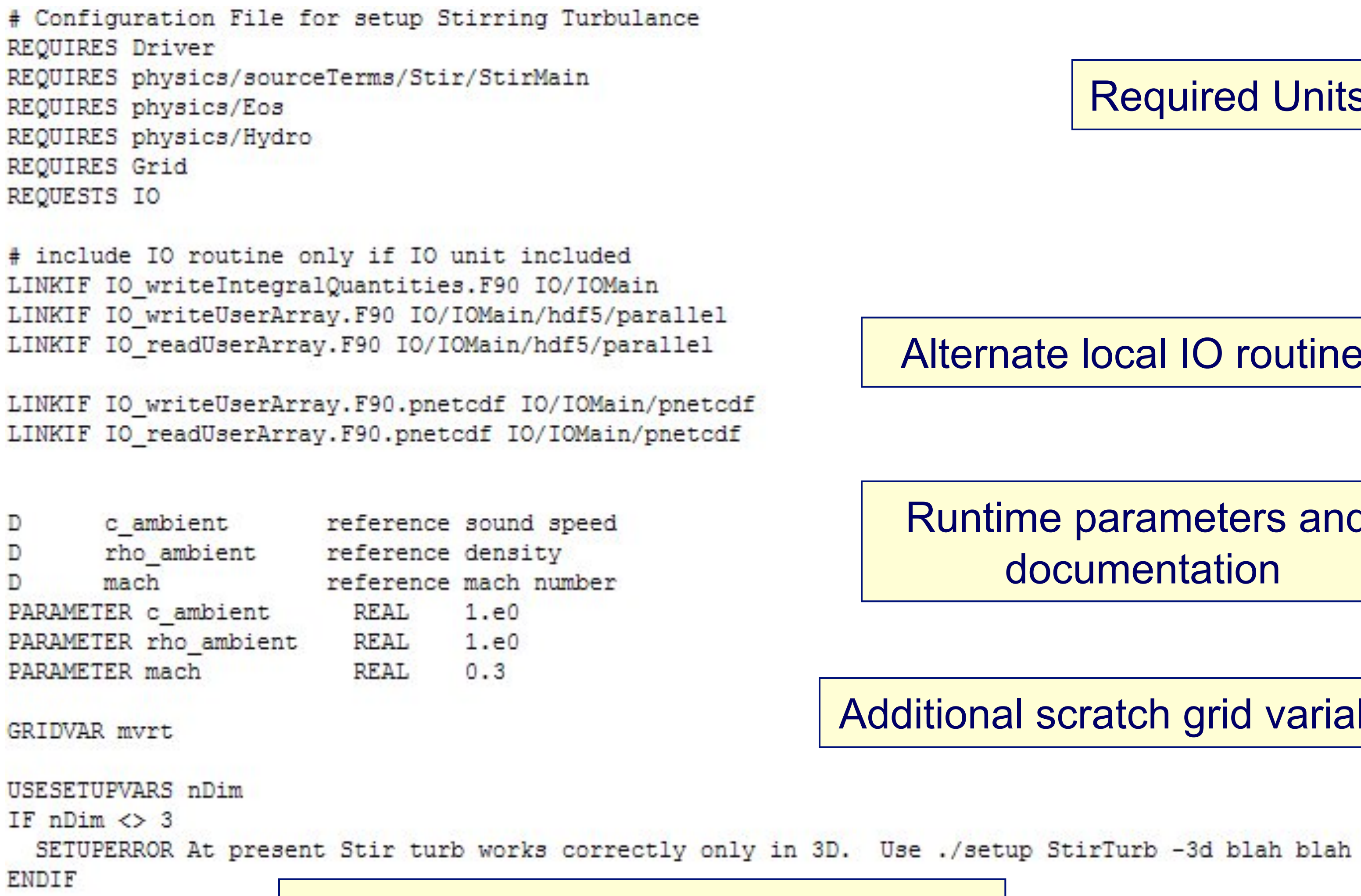
Python code links together needed physics and tools for a problem

- Traverse the FLASH source tree and link necessary files for a given application to the object directory
- Creates a file defining global constants set at build time
- Builds infrastructure for mapping runtime parameters to constants as needed
- Configures Makefiles properly
- Determine solution data storage list and create Flash.h
- Generate files needed to add runtime parameters to a given simulation.
- Generate files needed to parse the runtime parameter file.

Config file: Purpose

- Written in a FLASH-dependent syntax
- Needed in each Unit or Simulation directory
- Define dependencies at all levels in the source tree:
  - Lists required, requested, exclusive modules
- Declare solution variables, fluxes
- Declare runtime parameters
  - Sets defaults and allowable ranges – do it early!
  - Documentation – start line with “D”
- Variables, Units are additive down the directory tree
- Provides warnings to prevent dumb mistakes
  - Better than compiling and then crashing

Config file example



```
# Configuration File for setup Stirring Turbulance
REQUIRES Driver
REQUIRES physics/sourceTerms/Stir/StirMain
REQUIRES physics/Eos
REQUIRES physics/Hydro
REQUIRES Grid
REQUESTS IO

# include IO routine only if IO unit included
LINKIF IO_writeIntegralQuantities.F90 IO/IOMain
LINKIF IO_writeUserArray.F90 IO/IOMain/hdf5/parallel
LINKIF IO_readUserArray.F90 IO/IOMain/hdf5/parallel

LINKIF IO_writeUserArray.F90.pnetcdf IO/IOMain/pnetcdf
LINKIF IO_readUserArray.F90.pnetcdf IO/IOMain/pnetcdf


D      c_ambient       reference sound speed
D      rho_ambient     reference density
D      mach            reference mach number
PARAMETER c_ambient      REAL    1.e0
PARAMETER rho_ambient    REAL    1.e0
PARAMETER mach           REAL    0.3

GRIDVAR mvrt

USESETUPVARS nDim
IF nDim <> 3
  SETUPERROR At present Stir turb works correctly only in 3D.  Use ./setup StirTurb -3d blah blah
ENDIF
```

Required Units

Alternate local IO routines

Runtime parameters and documentation

Additional scratch grid variable

Enforce geometry or other conditions

Simple setup

```
hostname:Flash3> ./setup MySimulation -auto
```

setup script will automatically generate the object directory based on the MySimulation problem you specify

Sample Units File

```
INCLUDE Driver/DriverMain/TimeDep
INCLUDE Grid/GridMain/paramesh/Paramesh3/PM3_package/headers
INCLUDE Grid/GridMain/paramesh/Paramesh3/PM3_package/mpi_source
INCLUDE Grid/GridMain/paramesh/Paramesh3/PM3_package/source
INCLUDE Grid/localAPI
INCLUDE IO/IOMain/hdf5/serial/PM
INCLUDE PhysicalConstants/PhysicalConstantsMain
INCLUDE RuntimeParameters/RuntimeParametersMain
INCLUDE Simulation/SimulationMain/Sedov
INCLUDE flashUtilities/general
INCLUDE physics/Eos/EosMain/Gamma
INCLUDE physics/Hydro/HydroMain/split/PPM/PPMKernel
INCLUDE physics/Hydro/HydroMain/utilities
```

If you don’t use the -auto flag, you must have a valid Units file in the object FLASH directory (FLASH3/object/Units)

setup Shortcuts & help

- ./setup –help shows many fascinating options
- Shortcuts allows many setup options to be included with one keyword
- To use a shortcut, add +shortcut to your setup line
  - The shortcut ug is defined as:
    - ug:--with-unit=Grid/GridMain/:Grid=UG:
    - prompt> ./setup MySimulation -auto +ug
    - this is equivalent to typing in unit options with
    - -unit=Grid/GridMain/UG
    - -unit=IO/IOMain/hdf5/serial/UG (because the appropriate IO is included by default)
- Look in Flash3/bin/setup_shortcuts.txt for more examples and to define your own

Important Files Generated by setup

setup_call	contains the options with which setup was called and the command line resulting after shortcut expansion
setup_datafiles	contains the complete path of data files copied to the object directory
setup_defines	contains a list of all pre-process symbols passed to the compiler invocation directly
setup_flags	contains the exact compiler and linker flags
setup_libraries	contains the list of libraries and their arguments (if any) which was linked in to generate the executable
setup_params	contains the list of runtime parameters defined in the Config files processed by setup
setup_units	contains the list of all units which were included in the current setup
setup_vars	contains the list of variables, fluxes, species, particle properties, and mass scalars used in the current setup, together with their descriptions

Additional Files created by setup

- `Flash.h` contains
  - Problem dimensionality and size e.g. NDIM, MAXBLOCKS
  - Fixed block size dimensionality e.g. NXB, GRID_IJI_GC
  - Variable, species, flux, mass scalar numbers and list e.g. e.g. NSPECIES, DENS_VAR, EINT_FLUX
  - Possibly grid geometry GRID_GEOM
  - PPDEFINE variables showing which units are included e.g. FLASH_GRID_PARAMESH3

- `Simulation_mapIntToStr.F90, Simulation_mapStrToInt.F90`
  - Converts text strings to equivalent index in Flash.h e.g. “dens” maps to DENS_VAR=1

Architecture

Questions?