PROGRAMMING ABSTRACTIONS FOR ORCHESTRATION OF HPC SCIENTIFIC COMPUTING

ANSHU DUBEY
SETTING THE STAGE

What should my ideal computational tool do?

Everything really.

- Scan my brain
- Figure out what I want
- Scan the literature
- Figure out the equations
- Auto-generate the code
- Run it
- Analyze the data

I am happy to present the results.
Since programming models and other tools are not so obliging, let me reduce the complexity by several orders of magnitude.

If we were starting a new multiphysics exascale software project today, that expects to have long term use for scientific discovery, how should we design the software?

Chapel designers seem to think the way I do. I like the abstractions and the design, let me explain why.
THERE IS MORE

Validation

Implementation

Numerical solvers

Equations

Mesh/particles etc

Physical World

Domain expert

Model fidelity

Domain expert

Discretize

Model fidelity

Domain expert

Model

Applied Mathematician

Software Engineer, optimization experts

Verify accuracy stability

Performance

Applied Mathematician
ARCHITECTING SCIENTIFIC CODES

Desirable Characteristics

- **Extensibility**
  - Most use cases need additions and/or customizations

- **Portability**
  - Even the same generation platforms are different

- **Performance**
  - All machines need to be used well

- **Maintainability and Verifiability**
  - For credible and reproducible results
Desirable Characteristics

- **Extensibility**
  - Well defined structure and modules
  - Encapsulation of functionalities

- **Portability**
  - General solutions that work without significant manual intervention across platforms

- **Performance**
  - Spatial and temporal locality of data
  - Minimizing data movement
  - Maximizing scalability

- **Maintainability and Verifiability**
  - Clean code
  - Documentation
  - Comprehensive testing
ARCHITECTING SCIENTIFIC CODES

Why it is challenging

Extensibility
- Same data layout not good for all solvers
- Many corner cases
- Necessary lateral interactions

Portability
- Tremendous platform heterogeneity
- A version for each class of device => combinatorial explosion

Performance
- Solvers with low arithmetic intensity but hard sequential dependencies
- Proximity and work distribution at cross purposes

Maintainability and Verifiability
- Wrong incentives
- Designing good tests is hard
Design Approach

Taming the Complexity: Separation of Concerns

**Subject of research**
- Model
- Numerics

**Client Code**
- Mathematically complex

**Infrastructure**
- Data structures and movement

**More Stable**
- Discretization
- I/O
- Parameters

Treat differently

Hide from one another

Apply to both kinds

- Logically separable functional units of computation
- Encode into framework
- Differentiate between private and public
- Define interfaces
SEPARATION OF CONCERNS

- Requirements
- Software Architecture
  - API Design
- Implement
- Test
- Maintain
- Augment
- Model
- API
- Design
  - Develop
- Validate
- Integrate
DESIGN PHILOSOPHY

- Infrastructure design
  - Take time to discuss, iterate over requirements and specification
  - Keep end users involved
    - Not doing so leaves possible options on the table
- Simple is better
  - Flexibility Vs transparent to the user
    - Flexibility wins
- Hierarchical access to features
INTERACTION BETWEEN INFRASTRUCTURE AND PHYSICS

infrastructure

physics

Interfaces

Wrapper layer
Example Software: FLASH

- Cosmological cluster formation
- Supersonic MHD turbulence
- Rayleigh-Taylor instability
- Ram pressure stripping
- Type Ia SN
- Core collapse supernovae
- Vulcan laser experiments: B-field generation/amplification
- Accretion torus
- Laser slab
- Rigid body structure
Many components under research
Software continuously evolving
Compute on expensive, rare resources
All use cases are different and unique
FLASH CODE BASICS

- An application code, composed of encapsulated functional *units*.
  - Units are combined and composed to form applications
  - Not one monolithic binary, each problem has its own distinct binary
- Setup tool (python) parses Config files, picks specific implementations of units and composes full application
  - Units can have alternative implementations
    - Three implementations of mesh are supported
  - Composability implies any of the implementations can be picked
- Mostly Fortran, some C, about 1.5 million lines of code
- Portable, and until recently performance portable
DESIGN CONSIDERATIONS

- Encapsulation and interfaces
- Separation of concerns
- Extensibility
- Locality
- Composability
- Orchestration
- Cost accounting
ENCAPSULATION

- Virtual view of functionalities
- Decomposition into units and definition of interfaces
EXTENSIBILITY
ADD A UNIT

FLASH Driver
Grid API
GridMain Config
AMR Config
Implementation

namespace
organizational

FLASH Driver
Grid API
GridMain Config
AMR Config
Implementation

Hydro API
HydroMain Config
Unsplit Config
Implementation

Other units
Call Grid_initDomain
... (call other units)

unmodified

6/26/19
EXTENSIBILITY AND LOCALITY

ADD A SUBUNIT
COMPOSABILITY

- AMR infrastructure: refinement, load balancing, work redistribution
- Meta-information about domain sections
- Asynchronization at block and operator level
- No kernel optimization in this part
COMPOSITION

- Abstractions for performance portability
- Ability to express operations at a higher level
- Do away with optimization blockers

- Leave it to tools and compilers to optimize
CODE TRANSFORMATION

- Two different scopes
  - The usual one
    - Write code once, generate ”optimized” code for the target
    - Down at the level of loop nests or kernels
      - Best done for limited scope computations
    - We intend to use transpiler being developed by collaborators
    - Turns IR into constrained python, optimized code generated from there.
  - The not so usual one
    - High level orchestration of operators
    - Determined during application configuration
    - Communicated to the runtime in part
ORCHESTRATION SYSTEM

- Task composer – used for configuration
  - Extension of the original FLASH “Config” files
  - A configuration DSL
  - Encode meta-information for application construction in FLASH-specific syntax as needed

A primer on how FLASH framework configures application.
CONFIG FILES

- Can exist anywhere in the directory structure
- Encode all meta-information for that level
  - Unit dependencies
  - State variables needed
  - State variables that need reconciliation at fine-coarse boundaries
  - Runtime environment

REQUIRES Driver
REQUIRES physics/Hydro
REQUIRES physics/Eos/EosMain/Helmholtz
REQUIRES physics/sourceTerms/Burn/BurnMain/nuclearBurn
REQUIRES Simulation/SimulationComposition

PARAMETER xhe4 REAL 0.0 [0.0 to 1.0]
PARAMETER xc12 REAL 1.0 [0.0 to 1.0]
PARAMETER xo16 REAL 0.0 [0.0 to 1.0]
CONFIGURATION

- Hydro/MHD
  - Explicit
  - Stencils

- Self Gravity
  - Semi-implicit
  - Stencils, FFT etc

- Diffusion
  - Implicit

- Radiation
  - Implicit

- Laser Drive

- Shock Tube

- Library

- EOS
  - Pointwise
  - Table lookup

- Burn
  - Pointwise
  - ODE

- Particles
  - Lagrangian

Dubey et al, Parallel Computing 2009
CONFIGURATION

**Evolution (time stepping)**

- Hydro/MHD Explicit Stencils
- Self Gravity Semi-implicit Stencils, FFT etc
- Radiation Implicit
- Diffusion Implicit
- Cellular

- EOS Pointwise Table lookup
- Burn Pointwise ODE
- Particles Lagrangian
- Laser Drive

Dubey et al, Parallel Computing 2009
CONFIGURATION

Evolution (time stepping)

Hydro/MHD
Explicit
Stencils

GCD

Diffusion
Implicit

EOS
Pointwise
Table lookup

Burn
Pointwise
ODE

Particles
Lagrangian

Self Gravity
Semi-implicit
Stencils, FFT etc

Laser
Drive

Radiation
Implicit
CONFIGURATION

Evolution (time stepping)

Hydro/MHD
Explicit Stencils

Self Gravity
Semi-implicit Stencils, FFT etc

EOS
Pointwise Table lookup

Burn
Pointwise ODE

Particles
Lagrangian

Diffusion Implicit

Radiation Implicit

Laser Drive

HEDP

Library

Dubey et al, Parallel Computing 2009
COMPOSER FILES

- Same philosophy
- Keep them separate from Config files
  - More complex
  - Functionally different
  - Operate at individual unit level
- Build a separate tool
  - Could be a DSL compiler
    - We prefer to keep it simple
    - Time will tell if we can
- Parse the meta-information and produce executable code
OUR VISION

Task Composer

- Platform Information
- Solver Information
- Memory Requirements
- Kernels
- Operation

Emitted code
allocateMemoryHost()
allocateMemoryAccel()
movedata_1()
kernel_1()

...:
kernel_M()
movedata_M()
kernellN()
movedata_P()
deallocateMemoryHost()
deallocateMemoryAccel()
Task Composition: scheduleComputations(gpu={gcFill, computeFluxes, updateSoln, Eos}, cpu={computeDt}, moveDataBack=True)
BUILDING THE CODE

- Configuration in three stages
  - Stage 1 – the usual running of setup script
  - Stage 2 – run the task composer
  - Stage 3 – run the transpiler

- Run make as usual

- The orchestrator generated in the process
  - Launches various threads that control run time
  - May or may not interact with AMReX asynchronization

Lot of open questions still, but we believe that this is the right approach
WHY THIS WAY - PARALLELISM

- MPI is not difficult, decomposition is
- In parallelization neither all nor none is good
  - All – leave everything to the compiler
    - Domain specific knowledge lost – wasted opportunity
    - Compilers get impossible job, cannot optimize
  - None – orchestrate everything explicitly
    - Not feasible for even moderately complex application
    - Impossible from productivity perspective
- Whichever model is used, understanding the parallelizable structure of application is critical
- Constructs to encode the understanding needed
WHY THIS WAY - KERNELS

- C++ => Pushing a needlessly complex language that lacks basic structures
  - If there is a mesh there are 3D arrays
    - meta-data built and carried around
    - Explicit order of access and order of operations
  - No graceful way to encode lack of dependence
- Maintainable code in clean constructs, perhaps in python eventually
- We can also exploit alternative implementations at arbitrary granularity
ADVANTAGES

- All code can be compiled with standard compilers
- Constructs for expressing parallelism at different granularities
- Limit intelligence needed in any one tool
- Domain knowledge encoded in composer file, helps with optimizations

This is why I think Chapel designers think the way I do.
Questions ?