Architecting Community Codes

Anshu Dubey

August 10, 2015
ATPESC - Community Codes and Software Engineering Session
Outline

• Software Engineering for Scientific Software

• Constraints for Community Codes

• Architecting for Massive Parallelism and Heterogeneity

• Handling Legacy Codes
Software Process Components

• For All Codes
  – Code Repository
  – Build Process
  – Code Architecture
  – Coding Standards
  – Verification Process
  – Maintenance Practices

• If Publicly Distributed code
  – Distribution Policies
  – Contribution Policies
  – Attribution Policies

You will learn more about many of these topics tomorrow. Also visit https://ideas-productivity.org/resources/howtos/
Architecting Scientific Codes

• Desirable Features
  – Well defined structure and modules
  – Encapsulation of functionalities
  – Minimization of data movement
  – Maximization of locality and scalability
  – Portability

• Constraints
  – Accuracy and stability of numerics
  – Multiple solvers with diverse requirements
  – Intertwined interactions among solvers
  – Little or no duplication of expertise
Architecting Scientific Codes

• Why it gets messy
  – Well defined structure and modules
    • Same data layout not good for all solvers
    • Many corner cases (branches, other special handling)
  – Encapsulation of functionalities
    • Necessary lateral interactions
  – Minimization of data movement
    • Necessity of transposition / other form of copy
  – Maximization of locality and scalability
    • Solvers with low arithmetic intensity but hard sequential dependencies
    • Proximity and work distribution at cross purposes
Overarching Theme

• Differentiate between physical view and virtual view

• Simpler world view at either end enables separation of concerns

• Hard-nosed trade-offs
Example: PDE’s

Real view: A whole domain with many operators

Functional decomposition

Virtual view: domain sections as stand-alone computation unit

Memory access and compute optimization

Parallelization and scaling optimization

Virtual view collection of components

Customizations can be hidden under the virtual views as needed
Resources

https://www.cct.lsu.edu/research/cyber-advancement/cactus
http://flash.uchicago.edu/site/flashcode
https://computation-rnd.llnl.gov/SAMRAI
http://ambermd.org
https://www.earthsystemcog.org/projects/esmf
https://commons.lbl.gov/display/chombo


Code Verification

Many stages and types of verification

• During initial code development
  – accuracy and stability during development of the algorithm
  – matching the algorithm to the model
  – interoperability of algorithms

• In later stages
  – Ongoing maintenance
  – while adding new major capabilities or modifying existing capabilities
  – Preparing for production

• Mix of automation and human-intervention
Ongoing Maintenance

• The Selection of Tests
  – Highly composable code => too many configurations
  – Runtime parameters => variability in execution
  – Also need to verify transparent restart
  – For categorization of tests see https://ideas-productivity.org/resources/howtos/ideas-testing-definitions
    – Focus here on unit /no-change tests

• Running the tests
  – Must run on multiple platforms
  – Maximize coverage for functions and interoperability
  – Look for optimizations where possible
• Selecting Tests
  – One approach : use a matrix
  – Put infrastructure components in rows, science components in columns
  – List interoperability constraints, and pick apps
    • All unit tests
    • Tests for ongoing productions
    • Tests known to be sensitive to perturbations
    • Least complex tests that can cover the empty spots
    • Least complex tests that meet the missing interoperability constraints

• Running Tests
  – Select a test-harness frameworks (i.e. Jenkins)
  – Add selected tests, automate as much as possible
  – For more details see  
  – Example http://flash.uchicago.edu/site/testsuite
## Example: FLASH Tests Collection

<table>
<thead>
<tr>
<th>test type</th>
<th>approach</th>
<th>coverage</th>
<th>examples</th>
<th>done by</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit test</td>
<td>use alternative way to generate verification data</td>
<td>a capability or a solver</td>
<td>guard cells, particle integration</td>
<td>test-suite software</td>
</tr>
<tr>
<td>comparison test</td>
<td>against approved benchmark</td>
<td>interoperability among units and apps</td>
<td>advection, shock tube, rotor</td>
<td>test-suite software</td>
</tr>
<tr>
<td>restart test</td>
<td>against two approved benchmarks</td>
<td>transparent restart</td>
<td>advection shock tube rotor</td>
<td>test-suite software</td>
</tr>
<tr>
<td>target platform</td>
<td>manual verification</td>
<td>specific application</td>
<td>RTflame</td>
<td>human experts</td>
</tr>
<tr>
<td>benchmark update</td>
<td>manual verification</td>
<td>affected tests</td>
<td>solver upgrade</td>
<td>human experts</td>
</tr>
<tr>
<td>populating new test platform</td>
<td>combination of manual and automated</td>
<td>all tests</td>
<td>compiler upgrade</td>
<td>human experts and test suite software</td>
</tr>
</tbody>
</table>
Outline

• Software Engineering for Scientific Software

• **Constraints for Community Codes**

• Architecting for Massive Parallelism and Heterogeneity

• Handling Legacy Codes
Scientific Community Codes Have Followed Different Paths:

• The most common path
  – Someone wrote a very useful piece of code
  – Collaborations happened, critical mass of users achieved, code becomes popular
  – No focused effort, no software process, limited shelf life

• More sustainable path
  – Some long term planning might result in better engineered code
  – Thought given to extensibility and for future code growth
  – As the code grows so does its community supported model

• The desirable path
  – Explicit funding to support a design phase with expectation of longevity and good engineering
  – When it works outcome can go way beyond original expectations
Varying User Expertise

- Novice users – execute one of included applications
  - change only the runtime parameters
- Most users – generate new problems, analyze
  - Generate new Simulations with initial conditions, parameters
  - Write alternate and/or derived functions for specialized output
- Advanced users – Customize existing functions
  - Add small amounts of new code needed by their application
- Expert – new research
  - Completely new algorithms and/or capabilities
  - Can contribute to core functionality
Software Engineering

• Strong interfaces and encapsulation (enforced by the language or build system) required for community participation.
  – Users want to customize in many different ways
    • Depends somewhat on the code architecture
    • Add needed interfaces on top of infrastructure
    • Use derived classes
• No comprehensive in-house support for all features
• Transient developer and user population
• Users acquire a critical dependence for their work
  – Makes it harder to build the community
Distribution Policies

- The licensing agreement
  - How restrictive?
- Distribution control
  - Who can get the code
  - Should there be a registration requirement
- What is included in the release
  - The degree of support for released components
- How often to release
  - Trade-off between making capabilities available quickly and the overhead of releasing
Contribution Policies

• Balancing contributors and code distribution needs
  – Contributors want their code to become integrated with
    the code so it is maintained, but may not want it
    released immediately
    • Not exercised enough
    • Contributor may want some IP protection

• Maintainable code requirements
  – The minimum set needed from the contributor
    • Source code, build scripts, tests, documentation

• Agreement on user support
  – Contributor or the distributor

• Add-ons : components not included with the
distribution, but work with the code
Community Building

- Popularizing the code alone does not build a community
- Neither does customizability – different users want different capabilities

So what does it take?
- Enabling contributions from users and providing support for them
- Including policy provisions for balancing the IP protection with open source needs
- Relaxed distribution policies – giving collective ownership to groups of users so they can modify the code and share among themselves as long as they have the license

More inclusivity => greater success in community building
An investment in robust and extensible infrastructure, and a strong culture of user support is a pre-requisite
Outline

• Software Engineering for Scientific Software

• Constraints for Community Codes

• Architecting for Massive Parallelism and Heterogeneity

• Handling Legacy Codes
Example: PDE’s

Real view: A whole domain with many operators

Functional decomposition

Spatial decomposition

Virtual view: domain sections as stand-alone computation unit

Abstraction at solver level

Virtual view collection of components

Fusing Functions

Dynamic Scheduling

code transformation

Memory access and compute optimization

Parallelization and scaling optimization

Write solvers as independent tasks
Explicitly call out dependencies
Expose fusion possibilities

Dubey and Graves, A Design Proposal for a Next Generation Scientific Software Framework, HeteroPar 2015
Solver Level Abstractions

- **Stencil DSLs**
  - A stencil operator is a collection of shifts with corresponding coefficients
  - Applying the stencil operator
    - Weighted sum of some points on the mesh
    - Offset specified by the shift relative to the target

![Diagram of stencil operators with offsets <(-1,0), (1,0), (0,-1), (0,1), (0,0)>]
Hierarchical Decomposition

Real View

Virtual View

Tiling

- Hierarchy of tiling:
  - Larger non-overlapping tile maps to coherence domain
  - Smaller overlapping tile exposes more parallelism
  - Parameterize end-points, shape tiles as needed
Asynchronous Execution

• Barriers are the easy way to reconcile dependencies
  – Take away the option of pipelining and/or overlapping

• With hierarchical spatial and functional decomposition
  rich collection of tasks
  – Articulate dependencies explicitly
  – Let the framework find the unit of computation that is
    ready and hand it to client code with all the necessary data
    • Under the hood, framework can be managing dependencies
    • If client code assumes not-in-place update each of the tiles is a
      task with neighborhood dependencies

• Can be made into build or run environment
  specifications through appropriate parameterization
Putting it all Together

• The construction of operators
  – Express computation in the form of stencil operators or other appropriate abstraction
  – Specify the part of the domain, and the conditions under which the operators apply
    • Use masks to take care of branching

• Mix-mode parallelism
  – Parameters to control the degree of tiling or other forms of mix-mode parallelism
    • Could be handed to the compiler when technology arrives
  – Framework forms the data containers

• Dynamic tasking
  – Smarter iterators that are aware of mix-mode parallelism and dependencies
  – The iterating loops give up control and do while loops
Integrated Option: Kokkos

- Polymorphic multidimensional array (logical indexing)
  - Layout: multi-index (i,j,k,...) ↔ memory location
- Memory Space: where data resides
  - Differentiated by performance; e.g., capacity, latency, bandwidth
- Execution Space: where functions execute
  - Encapsulates hardware resources; e.g., cores, GPU, vector units, ...
  - Identify accessible memory spaces
- Execution Policy: how (and where) a user function is executed
  - E.g., data parallel range: concurrently call function(i) for i = [0..N)
  - User's function implemented as a C++ lambda or functor
- Pattern: parallel_for, parallel_reduce, parallel_scan, task-dag, ...
- Compose: pattern + execution policy + user function; e.g.,
  - Parallel pattern: foreach, reduce, scan, task-dag, ...
  - Parallel loop/task body: C++11 lambda or C++98 functor
  - parallel_pattern( Policy<Space>, Function);
- Execute Function in Space according to pattern and Policy
Some Other Options

• Many efforts to provide tools to application developers
  – TiDA, HTA: managing tiling abstractions
  – GridTools: comprehensive solution from CSCS-ETH
  – Dash: managing multilevel locality
  – Task based processing – OCR, charm++, HPX, Quark etc
  – Language based solutions – Julia, Chapel, UPC++ etc
  – Domain specific languages
Other Things to Consider

• Leverage existing software
  – Libraries may have better solvers
    • Off-load expertise and maintenance
  – Examine the interoperability constraints
    • Many times the cost is justified even if there is more data movement

• More available packages are attempting to achieve interoperability
  – See if a combination meets your requirements

• May be worthwhile to let the library dictate data layout if the corresponding operations dominate

Institute an extremely rigorous verification regime at the outset
Outline

• Software Engineering for Scientific Software

• Constraints for Community Codes

• Architecting for Massive Parallelism and Heterogeneity

• Handling Legacy Codes
Handling Legacy Codes

• Whether the code is worth refactoring
  – Greatly exercised, so robust and reliable
  – A great deal of model and algorithm knowledge encoded
  – No new revolution change in the models and/or algorithms likely to occur in foreseeable future
  – New models need to be added to increase fidelity, doing it from scratch would take far too long

• How much of the code is worth retaining

• When does the cost of backward compatibility become too high

• How closely is the data layout tied to the solvers
Handling Legacy Codes

• Good interdisciplinary interactions critical
  – Avoid pitfalls such as too little understanding of the idiosyncrasies
    • Many times what seems like bad code is there for a reason
  – Avoid upsetting the sensitivities of code owners
    • Need for co-operative design, possibly redesign of key components
  – At least one member of the team should be able to speak multiple domain languages

• Examine the inherent granularities in the modeling
  – They usually translate into components
  – Eliminate unnecessary lateral dependencies in the logical view
    • They will help separate dependencies arising as an artifact of implementation

• Apply the design ideas as outlined earlier in the lecture
Handling Legacy Codes

• Leverage existing software
  – Libraries may have better solvers
    • Off-load expertise and maintenance
  – Examine the interoperability constraints
    • Many times the cost is justified even if there is more data movement

• More available packages are attempting to achieve interoperability
  – See if a combination meets your requirements

• Automate repetitive tasks where possible

Institute an extremely rigorous verification regime at the outset
Handling Legacy Codes: Raja

• Encapsulate architecture-specific concerns through four co-operating features

• Data type encapsulation
  – Hides non-portable compiler directives, data attributes, etc.

• Traversal template & execution policy
  – Encapsulates platform-specific scheduling & execution and code-specific
  – iteration patterns (typically a limited number of patterns per code)

• IndexSet
  – Encapsulates iteration space partitioning & data placement

• C++ lambda function
  – Captures loop body without modification (essential for RAJA adoption in legacy code)
Publications and Workshops

• Journals
  – Parallel Computing
  – International Journal of High Performance Computing Applications
  – Journal of Parallel and Distributed Computing
  – Computational Science and Discovery
  – Concurrency – Practice & Experience
  – Software – Practice & Experience

• Workshops
  – WolfHPC
  – WSSSPE
  – SEHPSSSE