Scientific Software Design

Anshu Dubey
Argonne National Laboratory

Software Productivity and Sustainability track, ATPESC 2021

Contributors: Anshu Dubey (ANL), Mark C. Miller (LLNL)
License, Citation and Acknowledgements

License and Citation

• This work is licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0).
• The requested citation the overall tutorial is: David E. Bernholdt, Anshu Dubey, Rinku K. Gupta, and David M. Rogers, Software Productivity and Sustainability track, in Argonne Training Program on Extreme-Scale Computing (ATPESC), online, 2021. DOI: 10.6084/m9.figshare.15130590
• Individual modules may be cited as Speaker, Module Title, in Better Scientific Software tutorial...

Acknowledgements

• This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
• This work was performed in part at the Argonne National Laboratory, which is managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357.
• This work was performed in part at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
• This work was performed in part at the Lawrence Livermore National Laboratory, which is managed by Lawrence Livermore National Security, LLC for the U.S. Department of Energy under Contract No. DE-AC52-07NA27344.
• This work was performed in part at the Los Alamos National Laboratory, which is managed by Triad National Security, LLC for the U.S. Department of Energy under Contract No.89233218CNA000001
• This work was performed in part at Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-NA0003525.
HPC Computational Science Use-case

More Scientific Understanding

More Diverse Solvers

Higher Fidelity Model

More Hardware Resources
HPC Computational Science Use-case

More Scientific Understanding

More Hardware Resources

More Diverse Solvers

Higher Fidelity Model

Heterogeneous models

Distributed memory model

Platform complexity
HPC Computational Science Use-case

- More Scientific Understanding
- More Hardware Resources
- More Diverse Solvers
- Higher Fidelity Model

- Many components may be under research
- Software continuously evolves
- All use cases are different and unique
General Design Principles for HPC Scientific Software

Considerations
- Multidisciplinary teams
  - Many facets of knowledge
  - To know everything is not feasible
- Two types of code components
  - Infrastructure (mesh/IO/runtime …)
  - Science models (numerical methods)
- Codes grow
  - New ideas => new features
  - Code reuse by others

Design Implications
- Separation of Concerns
  - Shield developers from unnecessary complexities
- Work with different lifecycles
  - Long-lasting vs quick changing
  - Logically vs mathematically complex
- Extensibility built in
  - Ease of adding new capabilities
  - Customizing existing capabilities
General Design Principles for HPC Scientific Software

Design first, then apply programming model to the design instead of taking a programming model and fitting your design to it.
A Design Model for Separation of Concerns

Infrastructure

Requirements

Software Architecture API Design

Implement

Test

Maintain

Augment

Capabilities

Model

API

Design Develop

Validate

Integrate
The Running Example

Let's say you live in a house with exterior walls made of a single material of thickness, $L_x$. Inside the walls are some water pipes as pictured below.

You keep the inside temperature of the house always at 70 degrees F. But, there is an overnight storm coming. The outside temperature is expected to drop to -40 degrees F for 15.5 hours. Will your pipes freeze before the storm is over?
Problem Specification - Design Considerations

• Specification
  – Solve heat equation with some initial and boundary conditions
  – Apply different integration methods

• What is infrastructure here?
  – Discretization/ State
  – Verification
  – I/O
  – Application of initial conditions
  – Runtime parameters
  – Comparison

• What is model here?
  – Initial conditions
  – Boundary conditions
  – Integration
Infrastructure API

• `process_args`(int argc, char **argv)
• static void `initialize`(void)
• void `copy`(int n, double *dst, double const *src)
• void `write_array`(int t, int n, double dx, double const *a)
• void `set_initial_condition`(int n, double *a, double dx, char const *ic)
Numerics API

- double l2_norm(int n, double const *a, double const *b)
- bool update_solution_crankn(int n, double *curr, double const *last, double const *cn_Amat, double bc_0, double bc_1)
- bool update_solution_upwind15(int n, double *curr, double const *last, double alpha, double dx, double dt, double bc_0, double bc_1)
- bool update_solution_ftcs(int n, double *uk1, double const *uk0, double alpha, double dx, double dt, double bc0, double bc1)
- void compute_exact_solution(int n, double *a, double dx, char const *ic, double alpha, double t, double bc0, double bc1)
Example: Architecting Multiphysics PDEs

- Virtual view of functionalities
- Decomposition into units and definition of interfaces

Spatial decomposition -> Virtual view: domain sections as stand-alone computation unit -> Parallelization and scaling optimization

Real view: A whole domain with many operators

Functional decomposition -> Virtual view collection of components -> Memory access and compute optimization
Example: Multiphysics PDEs for Distributed Memory Parallelism

- Virtual view of functionalities
- Decomposition into units and definition of interfaces

Spatial decomposition → Virtual view: domain sections as stand-alone computation unit → Parallelization and scaling optimization → Implemented by software and performance engineers

Real view: A whole domain with many operators → Functional decomposition → Virtual view: collection of components → Memory access and compute optimization → Implemented by domain experts and applied mathematicians

Implemented by domain experts and applied mathematicians

Implemented by software and performance engineers

 Example: Multiphysics PDEs for Distributed Memory Parallelism

Virtual view of functionalities

Decomposition into units and definition of interfaces
Example: Design for Extensibility from FLASH, Now Flash-X

Assumed that capabilities will be added for better models

- Assembly from components
- Decentralized maintenance of metadata
- Python tool to parse and configure
- OOP implemented through Unix directory structure and configuration tool

Key idea is distributed intelligence
Takeaways Until Now

- Differentiate between slow changing and fast changing components of your code
- Understand the requirements of your infrastructure
- Implement separation of concerns
- Design with portability, extensibility, reproducibility and maintainability in mind
- Do not design with a specific programming model in mind
A New Paradigm Because of Platform Heterogeneity

- Question - do the design principles change?

Software complexity

Platform complexity

Heterogeneous models
A New Paradigm Because of Platform Heterogeneity

- Question - do the design principles change?
- The answer is – not really
- The details get more involved
A Design Model for Separation of Concerns

Infrastructure

Requirements

Software Architecture API Design

Implement

Test

Maintain

Augment

Capabilities

Model

API

Design Develop

Validate

Integrate

This is where maximum change is likely
Design Guidance For Performance Portability

- Design for Hierarchical parallelism
- Design towards several thousand threads
- Design for a hierarchical memory space
- Design patterns that count, allocate, and reuse memory
- Avoid exposing/using non-portable vendor-specific options
Features and Abstractions that must Come in

Real view: A whole domain with many operators

Spatial Decomposition Blocks/tiles

Virtual view: domain sections as stand-alone computation unit

Load Distribution

Offloading and scaling optimization

Functional decomposition

Virtual view collection of components

Runtime management

Abstraction at solver level

code transformation

Memory access and compute optimization

IDEAS productivity
Features and Abstractions that must Come in

**How do abstraction layers work**
- Infer the structure of the code
- Infer the map between algorithms and devices
- Infer the data movements
- Map computations to devices
- These are specified either through constructs or pragmas

**Performance depends upon how well the mapping is done.**
Underlying Ideas

Make the same code work on different devices

• A way to let compiler know that "this" expression can be specialized in many ways
• Definition of specializations

Template meta-programming in abstraction layers
Underlying Ideas

Make the same code work on different devices

- A way to let compiler know that "this" expression can be specialized in many ways
- Definition of specializations

Template meta-programming in abstraction layers

Assigning work within the node

- "Parallel For" or directives with unified memory
- Directives or specific programming model for explicit data movement

More complex data orchestration system for asynchronous computation
Underlying Ideas

Make the same code work on different devices
- A way to let compiler know that “this” expression can be specialized in many ways
- Definition of specializations

Template meta-programming in abstraction layers

Assigning work within the node
- “Parallel For” or directives with unified memory
- Directives or specific programming model for explicit data movement

More complex data orchestration system for asynchronous computation

Look at what is needed, design for commonalities.
Underlying Ideas

Make the same code work on different devices
- A way to let compiler know that "this" expression can be specialized in many ways
- Definition of specializations

Template meta-programming in abstraction layers

Assigning work within the node
- “Parallel For” or directives with unified memory
- Directives or specific programming model for explicit data movement

Look at what is needed, design for commonalities.

Even when using third party abstraction tools understanding the code’s structure and needs is critical for performance portability.

More complex data orchestration system for asynchronous computation
Even when using third party abstraction tools understanding the code’s structure and needs is critical for performance portability … that translates to investing in design.

Make the same code work on different devices

- A way to let compiler know that "this" expression can be specialized in many ways
- Definition of specializations

Template meta-programming in abstraction layers

Assigning work within the node

- “Parallel For” or directives with unified memory
- Directives or specific programming model for explicit data movement

Look at what is needed, design for commonalities.

More complex data orchestration system for asynchronous computation
Final takeaways

• The key to both performance portability and longevity is careful software design
• Extensibility should be built into the design
• Design should be independent of any specific programming model
• Composability and flexibility help with performance portability

• Resources:
  – https://www.exascaleproject.org/
  – https://doi.org/10.6084/m9.figshare.13283714.v1
  – https://www.exascaleproject.org/event/kokkos-class-series

IDEAS productivity ECP