LA-UR-20-30197
Approved for public release; distribution is unlimited.

Title: THE RISTRA PROJECT: FY20/21 MILESTONE REPORT

Author(s): Daniel, David John; Hungerford, Aimee L.; Bergen, Benjamin Karl; Bowen, Dennis Brent; Burke, Timothy Patrick; Campbell, Joann Marie; Certik, Ondrej; Charest, Marc Robert Joseph; Chiravalle, Vincent P.; Davis, Erin Jessica; Demeshko, Irina P.; Dolence, Joshua C.; Drayna, Travis William; Dunning, Daniel Jeffrey; Edelmann, Philipp Valentin Ferdinand; Ferenbaugh, Charles Roger; Fryer, Christopher Lee; Garimella, Rao Veerabhadra; Grosset, Andre Vincent Pascal; Halverson, Scot Alan; Hammer, Hans Ruediger; et al.

Intended for: Report

Issued: 2020-12-11

THE RISTRA TEAM

# THE RISTRA PROJECT: FY20/21 MILESTONE REPORT

FOR ASC ATDM CDA LEVEL 1 AND ECP MILESTONES

LOS ALAMOS NATIONAL LABORATORY
LA-UR-20-XXXXX, DECEMBER 1, 2020

**The Ristra team**

**Co-leads**

Aimee Hungerford <aimee@lanl.gov>, Physics and Applications
David Daniel <ddd@lanl.gov>, Computer Science

**Contributors** (from Ristra, FleCSI, Portage, Legion, Kitsune, Lynx, SimTools and EAP teams)

Benjamin Bergen, Dennis Bowen, Timothy Burke, Joann Campbell, Ondrej Certik, Marc Charest, Vincent Chiravalle, Erin Davis, Irina Demeshko, Joshua Dolence, Travis Drayna, Daniel Dunning, Philipp Edelmann, Charles Ferenbaugh, Christopher Fryer, Rao Garimella, Andre Grosset, Scot Halverson, Hans Hammer, Angela Herring, Stuart Herring, Austin Isner, Christoph Junghans, Timothy Kelley, Evgeny Kikinzon, Oleg Korobkin, Brendan Krueger, Ricardo Lebensohn, Hyun Lim, Li-Ta Lo, Julien Loiseau, Peter Maginot, Mounia Malki, Christopher Malone, Len Margolin, Patrick McCormick, Michael McKay, Zachary Medin, Jacob Moore, Nathaniel Morgan, EunJung Park, HyeongKae Park, Robert Pavel, Katherine Perry-Holby, Jonathan Pietarila Graham, Nirmal Prajapati, Hoby Rakotoarivelo, Navamita Ray, Jonathan Regele, Andrew Reisner, Michael Rogers, Mikhail Shashkov, Daniel Shevitz, Galen Shipman, George Stelle, Karen Tsai, Jan Velechovsky, Daniele Versino, Alexander White, Ann Wills, Ryan Wollaeger, Jonathan Woodring, Nathan Woods, Duan Zhang

With apologies for omissions.

Los Alamos report: LA-UR-20-XXXXX, December 1, 2020

*Contents*

**Part I**

# Milestone completion

The metrics that we will use to demonstrate milestone completion, as presented to the committee in December 2019 and with revisions based on feedback, are as follows:

*Mission impact*

✓ Turn-around times for demonstration problem on 25% Sierra and 50% Trinity along with same for EAP codes on 50% Trinity
✓ Demonstration of ease of geometry setup, workflow, *etc.*
✓ *In situ* visualization and analysis enabled in *FleCSI*
✓ Integration of necessary physics
✓ Demonstration of mechanism for model-form uncertainty

*Portability*

✓ *Symphony* demonstrated on Sierra, Trinity and Astra. Performance and memory usage documented
✓ *Symphony* demonstrated with *MPI* and *Legion* runtimes. Performance and memory usage documented.
✓ Code differences for different systems and runtimes will be documented (Physics code changes; Infrastructure changes)

*Developer productivity*

✓ Ease of improving, modifying or extending code (Physics and CS Infrastructure)
✓ Examples of algorithmic diversity
✓ Code reuse between codes measured and demonstrated
✓ Software environment metrics (Repository statistics; Continuous integration statistics)
✓ Documentation and training materials enumerated

*Code performance*

✓ Time to solution, Grind times, I/O times, Efficiency
✓ Strong scaling (Trinity vs Sierra vs Astra; *MPI* vs *Legion*)
✓ Weak scaling (Trinity vs Sierra vs Astra; *MPI* vs *Legion*)
✓ Abstraction overhead, e.g. *FleCSI, Legion, Kokkos*

In this section we provide summary evidence for each check mark, referring to more detailed evidence either in the document below, or in the accompanying slide deck.

## 1 *Mission impact*

### *Turn-around times for demonstration problem on 25% Sierra and 50% Trinity along with same for EAP codes on 50% Trinity*

We have run a reference demonstration problem on 50% of Trinity KNL with both Ristra and EAP codes We have run the same on 25% Sierra and 100% Astra with Ristra codes. Time to solution was chosen at a reference simulation time based on physics exhibited Each run was optimized for turn-around time using features available in each code base Mesh choices (EAP is AMR; Ristra is unstructured), solver options, and timestep controllers were judiciously selected by each team. Turn-around for EAP and Ristra codes are comparable at 625 nodes, but currently Ristra codes do not weak scale as well as EAP codes (see Code Performance below) At smaller node counts, turn around time is comparable. For further analysis, including memory usage, see the addendum to the report available to the committee in the Wednesday review session.

| Platform | Nodes | Cycle time (s) | Time to solution (s) |
|---|---|---|---|
| Trinity Ristra | 5000 | 33.55 | 28214 |
| Trinity EAP | 5000 | 41.35 | 13397 |
| Sierra Ristra | 1096 | 12.10 | 12099 |

Table 1: Turn around times for the demonstration problem.

### *Demonstration of ease of geometry setup, workflow,* etc.

A number of tools have been developed within *Ristra* or integrated into the workflow for *Ristra* codes. These include *TIDE,* a Lua-based compile- and run-time problem setup tool, and *Crosslink/Parmesh,* a scalable, parametric meshing tool. These are demonstrated in subsection *The physics developer experience* and in the accompanying slide deck. Examples of integration with the Common Modeling Framwork (*CMF)* are also included.

This work is further covered in talks from Chris Malone and Aimee Hungerford on physics developer and user experiences in session 2 of the review. User experiences were gathered from potential code users tutorial sessions on ICF problems with Symphony and positive initial feeback was received:

> "Overall, for the first time seeing ... symphony, I found the decks pretty straightforward to follow/track, as well as to modify. ... The flexibility and ease of switching between physics models was great, particularly the hydro. 3D run times were faster than I anticipated..."

> "It was very easy to take the 3D ICF input deck and make it into a 2D input with a simple perturbation of the intial conditions. Even never

having heard of Lua before . . . it took about 2 minutes to make the conversion from 3D to 2D (worked the first time), and maybe 5 minutes to impose a sinusoidal density perturbation into the ice and ablator materials."

### In situ *visualization and analysis enabled in* FleCSI

*In situ* visualization using *Paraview/Catalyst* has been available in *FleCSI*-based codes since 2016 and has recently been enabled and demonstrated in *Symphony*

### *Integration of necessary physics*

Integration of the targeted physics has been completed in *Ristra* codes and will be demonstrated in the addendum to the report available to the committee in the Wednesday review session.

### *Demonstration of mechanism for model-form uncertainty*

A number of physics model swaps are demonstrated in the accompanying slide deck in the "User experience" section.

## 2 *Portability*

### Symphony *demonstrated on Sierra, Trinity and Astra. Performance and memory usage documented.*

*Symphony* has successfully run a demonstration multi-physics problem on 25% Sierra, 50% Trinity KNL and 100% Astra.

### Symphony *demonstrated with* MPI *and* Legion *runtimes. Performance and memory usage documented.*

*Symphony* is portable to both *MPI* and *Legion* parallel backends and has run with both on a variety of platforms including *Sierra*. Strong scaling results are presented in the "CS developer experience" section and slides. Weak scaling is a work in progress as we await the availability of a new release of *Legion* that addresses scalability challenges triggered by unstructured-mesh *FleCSI*-based codes.

#### CODE DIFFERENCES FOR DIFFERENT SYSTEMS AND RUNTIMES WILL BE DOCUMENTED

This is documented in the accompanying slides.

## 3 *Developer productivity*

### *Ease of improving, modifying or extending code (Physics and CS Infrastructure)*

The modular architecture of *FleCSI*-based codes with well-defined data and execution models ease the path to extending methods and algorithms, while the separation of concerns between physics and CS issues allows for more efficient development across the team. These benefits are discussed in the *Productivity in the Ristra Environment* section. In particular, the Multi-Material Multi-Physics (M3P) discipline developed for *FleCSI*-based codes and described in the "Physics developer experience" greatly eases physics implementation.

### *Examples of algorithmic diversity*

This is captured throughout the report and accompanying slides.

### *Code reuse between codes measured and demonstrated*

Described in the accompanying slides and in the "Physics developer experience" section.

### *Software environment metrics*

These are captured in the accompanying slides.

### *Documentation and training materials enumerated*

- *FleCSI* documentation: `https://laristra.github.io/flecsi/`
- *Portage* documentation: `https://laristra.github.io/portage/`
- *Friendly User Cookbooks* have been developed for *Symphony* and *FleCSALE-mm* as part of early engagement with users.

## 4 *Code performance*

### *Performance*

### *Time to solution, cycle times, etc.*

### *Weak scaling*

*Trinity, Sierra, Astra* A weak scaling study of a demonstration problem has been performed on up to 50% Trinity KNL, 25% Sierra, and 100% Astra. simulation, see Figure 1.

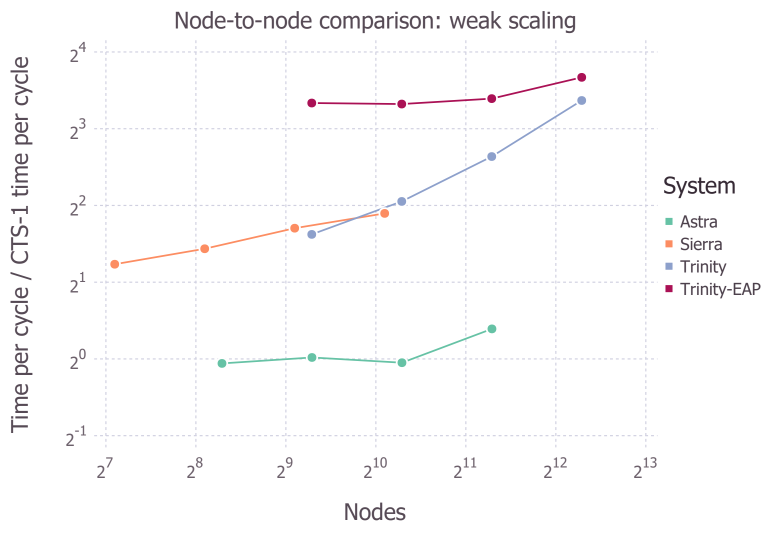


Figure 1: Weak scaling of the demonstration problem, including runs at the target scales of 50% Trinity KNL, 25% Sierra, and 100% Astra. Numbers are normalized to CTS-1 performance at 625 nodes. Notes: (a) The Astra line is from an earlier simulation time since that is where we had a consistent set of data, and hence is looks better; (b) The EAP time per cycle is higher, but for algorithmic reasons they need fewer cycles their time to solution is better.

### *Strong scaling*

Strong scaling results for the demonstration problem on Sierra and Astra are shown in Figure 2.

For Trinity KNL a full set of strong scaling data at the target scales is not available at the largest scales but we have collected data at smaller scales, as shown in Figure 3.

A discussion of throughput comparing Ristra and EAP codes can be found in addendum to the report that will be available to the committee in the Wednesday session of the review.

### *MPI vs Legion*

At present we are unable to run our full-physics codes at large scale using *legion* because they trigger bad scaling of *Legion*'s data-alias equivalence sets. A solution from the *Legion* team is in hand but not yet demonstrated.

This prevents us from a meaningful comparative study of weak scaling our demonstration problem.

However, a smaller scales (and with a correspondingly smaller mesh) we are able to explore strong scaling behaviour as shown in Figure 4.

### *Abstraction overhead, e.g.* FleCSI, Legion

In general FleCSI imposes negligible overhead relative to the underlying runtime Anecdotal evidence: using a code with both FLeCSI/MPI

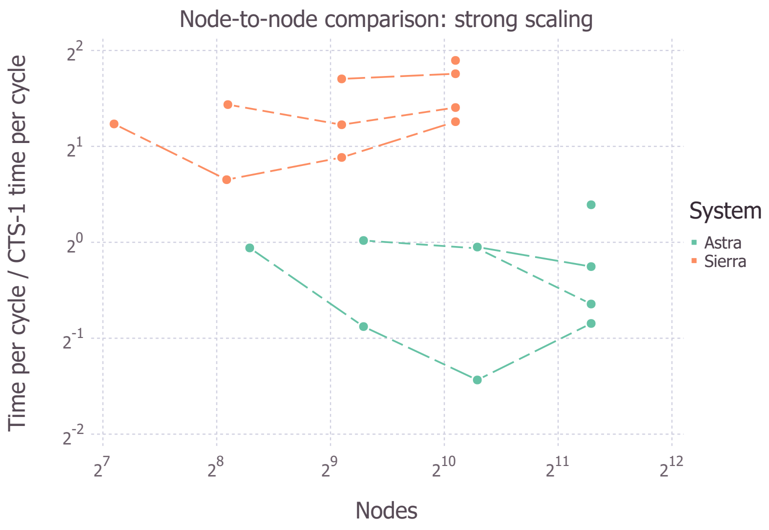


Figure 2: Work in progress for strong scaling plots for Sierra and Astra for the demonstration problem. Numbers are normalized to CTS-1 performance at 625 nodes. The different lines for each system are: (a) single point: full mesh; (b) long dash; half mesh; (c) short dash: quadrant mesh; (d) short long dash: octant mesh.

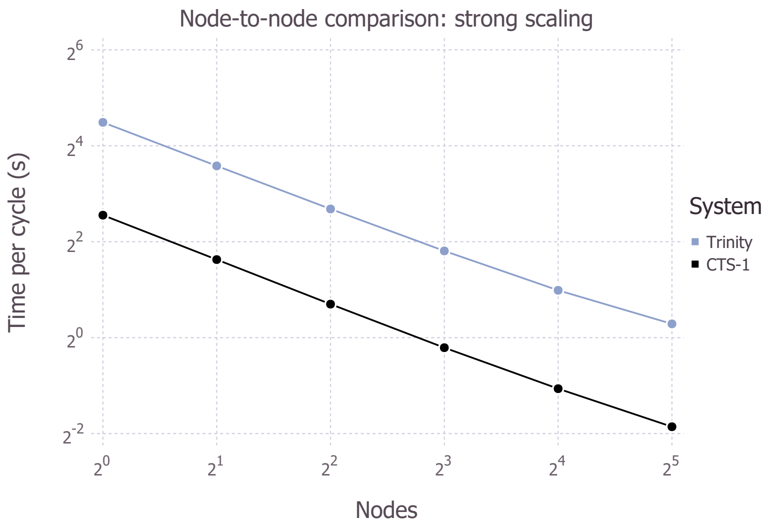


Figure 3: A strong scaling comparison between Trinity KNL and CTS-1 for a demonstration problem at smaller scales.

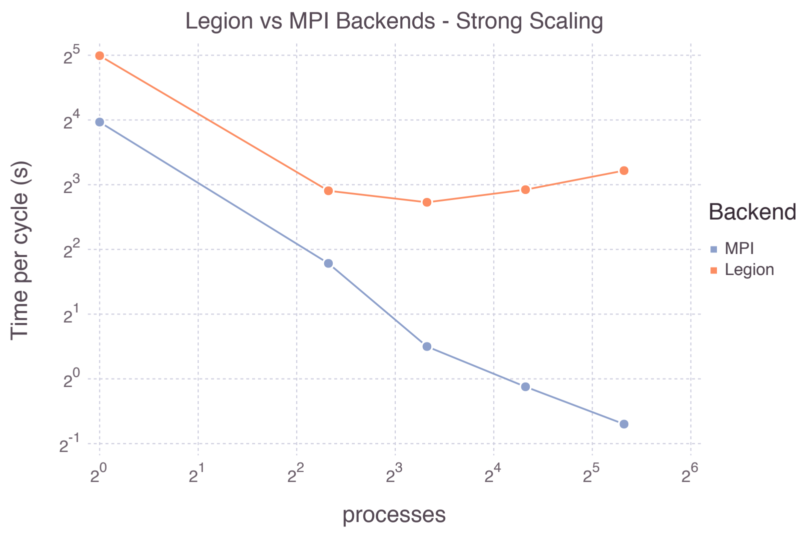


Figure 4: Legion vs MPI strong scaling of the demonstration problem on a small mesh on Sierra

and stand alone MPI running a realistic 5-material compression problem on CTS-1, no discernable difference in run time is observed Exceptions have been identified related to related to the implementation of certain mesh iterators in FleCSI 1.X: fixed in FleCSI 2.0 See "FleCSI" discussion.

For *Legion* we are still learning how to write applications in a data- and task-oriented fashion. For our simple test applications *Legion* can often be a factor of 2 slower in an initial implementation. Progress towards resolving this through appropriate task granularity and grouping is discussed in the "Legion and task parallelism" section of the report. Hierarchical tasks are a potential path towards backend-agnostic performance of the tasking model are available for us to explore in *FleCSI 2.0*.

**Part II**

# Introduction to *Ristra*

The ASC Advanced Technology Development and Mitigation (ATDM) sub-program was established in 2014 to develop new simulation tools operating on exascale-class computers to serve NNSA (see Appendix B).

Over the course of ATDM, LANL management have set a strategy for exascale-class application codes that follows two supportive and mutually risk-mitigating paths: evolution for established production integrated design codes (IDCs) – with a strong pedigree within the user community – based upon existing programming paradigms (MPI+X); and a new start ATDM project, *Ristra*, a high-risk/high-reward push for a next-generation multi-physics, multi-scale simulation toolkit based on emerging advanced programming systems (with an initial focus on data-flow task-based models exemplified by Legion). The role of *Ristra* as the high-risk/high-reward path for LANL's codes was fully consistent with the goals of ATDM as described in Appendix B, in particular its emphasis on evolving ASC capabilities through novel computing programming models and computing technologies.

In short, LANL has taken the opportunity provided by ATDM to invest in a long-term approach, recognizing that an incremental improvement of our current production IDC capabilities is insufficient to address new and complex challenges that LANL will face across its mission space in future, even though our production IDCs play a crucial role in maintaining the deterrence of the nation's stockpile and will continue to do so for at least a decade.

## 1 *Timeline*

LANL's ATDM CDA project *Ristra* started in FY15 and built off of the recommendations from an interdisciplinary task force that was formed in FY14. Initial high level goals and requirements were identified in the early months of FY15. A 5 year plan was presented that included a 2 year exploratory phase where the team surveyed available technologies and identified gaps through prototype code development activities. This exploratory phase completed with a FY16 Level 2 milestone.

The FY16 milestone delivered a plan for the following years of the project. By this time, sponsors from NNSA had extended the lifetime of the project, so the plan outlined work for the following 4 years (FY17-FY20). First, focus was on delivering a co-designed release of a high-level computer science abstraction layer *FleCSI* (flexible computational science infrastructure). Attention was placed on getting the design for *FleCSI*'s data model right. At the same time, *Ristra* recognized the need for its products to work together with current IDCs, so the creation of *Portage,* an extensible remap and link library was prioritized, with the expectation that this would provide early mission impact from the project. Low energy density (LED) physics implementations (i.e. multi-material hydro with solid mechanics) are known for the complexity of their data expression so LED multiphysics codes (*FleCSALE* and *FUEL*) were the first software products co-designed with *FleCSI* (FY17). The focus then shifted to a design of *FleCSI*'s task execution model. High energy density (HED) physics applications place stringent requirements on a task execution model in order to follow the flow of a multi-physics calculation. A HED radiation-hydrodynamics code *Symphony* was the resulting software product from the FY18 co-design efforts. The FY19 Level 2 milestone deliverable focused on the integration of these individually co-designed components (*FleCSALE*, *FUEL* and *Symphony*, with *Portage* for remap). In FY20, product integration continued and demonstrations on Advanced Technology Systems (ATS) became an emphasis, with performance optimization emphasized for delivery of a final high energy density simulation result for the current Level 1 milestone review.

The COVID-19 pandemic severely disrupted progress towards the FY20 goals, dramatically reducing the team's access to classified resources and negatively affecting options for demonstrating mission impact. Less tangible consequences of the current work environment are the sustained stress on staff and challenges to communicate effectively across a very diverse project. Both have undoubtedly reduced the productivity of the team as a whole.

As the ATDM initiative draws to a close, LANL management are currently developing a major new program and code strategy, and the foundations laid in *Ristra* are playing a significant role in those planning activities.

A more detailed timeline for the project is included in Appendix D.

## 2 *Goals*

The guiding philosophy behind the *Ristra* project was to create a software ecosystem that allows LANL the agility to build codes that respond to our ever changing mission questions and the ever changing HPC technologies.

Today's multi-physics codes are architected to support a specific multi-physics algorithm-set, with some ability to shift between physics models, but very limited ability to change the over-arching model for how the multi-physics components interact with one another. Similarly, they are designed with one mesh topology assumption and one parallel runtime assumption. These assumptions are hard-coded in to the implementation at all layers. Any shift in these assumptions would likely require a rewrite of the code base.

In contrast, *Ristra*, is developing a toolkit to enable the development of a set of next-generation multi-physics codes in diverse application domains spanning the NNSA mission space. We are targeting a combination of developer productivity, portability, and performance that will ease the job of developing and maintaining efficient codes for exascale-class computers and beyond. The name *Ristra* (meaning "string" in Spanish, but in a New Mexican context, a string of chiles), was chosen to emphasize that our aim was not to develop a single monolithic code, but rather a toolkit connecting a consistent suite of codes for different application domains.

*Ristra*, has taken a high-risk path that enables exploration of next-generation physics methods implemented using novel computer science technologies, with co-designed abstractions that enforce a separation of concerns between the two disciplines. The benefit of this path is greater agility, both in the code's ability to incorporate new algorithms (physics and computer science) and respond to the anticipated increased diversity in application requirements.

Other goals for the project (TBD)

- Task based design
- Exploration of novel programming paradigms
- A modular architecture where emphasizing open source components and code reuse across applications
- Code development processes designed to leverage solutions from the world-wide HPC community
- ...

## 3 Ristra *software architecture: planning for flexibility in a volatile future*

Two key challenges on the path to multi-physics computing at exascale and beyond are (a) abstracting details of underlying hardware and systems software from multi-physics code development and (b) solving mission-relevant problems at multiple physical scales.

To address the first challenge, *Ristra* is developing a computer science interface (*FleCSI*) that limits the impact of disruptive computer technology on the development of multi-physics codes. *FleCSI* enables the adoption of novel programming models and data management methods to address the challenges and diversity of new technology.

Simultaneously, *Ristra* is exploring the use of multi-scale numerical methods that offer improved physics fidelity and computing efficiency, in both high energy density (e.g. inertial confinement fusion), and low energy density (e.g. advanced materials and solid dynamics) regimes, that are relevant to NNSA's mission of stockpile stewardship.

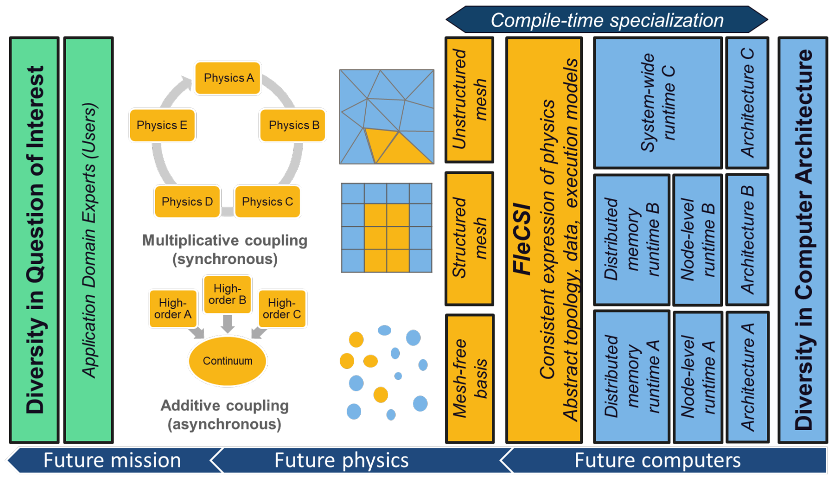

Compile-time specialization
Diversity in Question of Interest
Application Domain Experts (Users)
Physics A
Physics B
Physics C
Physics D
Physics E
Multiplicative coupling (synchronous)
High-order A
High-order B
High-order C
Continuum
Additive coupling (asynchronous)
Unstructured mesh
Structured mesh
Mesh-free basis
FleCSI
Consistent expression of physics
Abstract topology, data, execution models
System-wide runtime C
Distributed memory runtime B
Node-level runtime B
Distributed memory runtime A
Node-level runtime A
Architecture C
Architecture B
Architecture A
Diversity in Computer Architecture
Future mission
Future physics
Future computers

Figure 5: *Ristra* multi-physics code architecture

*Ristra* has adopted a software architecture (Figure 5) that emphasizes a separation of concerns between the computational physicists who need to respond to a diversity in questions of interest, and computer scientists who need to adapt to changing software and hardware computing technologies. That separation of concerns is implemented through an open-source abstraction layer, *FleCSI*. *FleCSI* can be thought of as a toolkit for building discretizations and physics operators to support physics expression over an abstract data and execution model that can target a variety of underlying parallel runtimes ranging from traditional MPI implementations to more capable modern runtimes like the *Legion* programming system that originated at Stanford and is being actively developed at NVIDIA and Los Alamos.

The *FleCSI* execution model is an asynchronous task-based programming model that is well suited to highly heterogeneous computer architectures, and is also well suited to explorations of new more asynchronous algorithmic techniques. Specifically, we are exploring novel multi-physics couplings in our codes, including a number of multi-scale algorithms that break the conventional operator split implementations of current multi-physics codes into a more concurrent model that has potential benefits for execution on extreme-scale computer platforms.

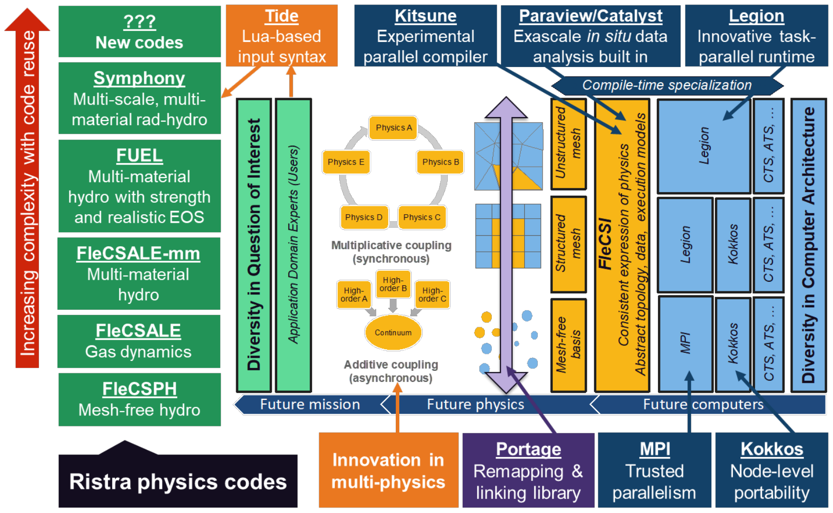

Increasing complexity with code reuse
??? New codes
Symphony Multi-scale, multi-material rad-hydro
FUEL Multi-material hydro with strength and realistic EOS
FleCSALE-mm Multi-material hydro
FleCSALE Gas dynamics
FleCSPH Mesh-free hydro
Ristra physics codes
Tide Lua-based input syntax
Kitsune Experimental parallel compiler
Paraview/Catalyst Exascale in situ data analysis built in
Legion Innovative task-parallel runtime
Compile-time specialization
Diversity in Question of Interest
Application Domain Experts (Users)
Physics A
Physics B
Physics C
Physics D
Physics E
Multiplicative coupling (synchronous)
High-order A
High-order B
High-order C
Continuum
Additive coupling (asynchronous)
Unstructured mesh
Structured mesh
Mesh-free basis
FleCSI
Consistent expression of physics
Abstract topology, data, execution models
Legion
MPI
Kokkos
CTS, ATS, …
Diversity in Computer Architecture
Future mission
Future physics
Future computers
Innovation in multi-physics
Portage Remapping & linking library
MPI Trusted parallelism
Kokkos Node-level portability

Figure 6: *Ristra*: state of the project

In a multi-physics code, we need a variety of types of discretization – structured mesh, unstructured mesh, and particle-based, and so on – to operate together, so an important part of our project is *Portage*, a remap and link library that allows physical geometry and data to be mapped between discrete representations (see Figure 6). *Portage* has been developed as a stand-alone library that can be readily extended to provide link capabilities between arbitrary computational physics codes, but is at the same time sufficiently lightweight to serve as an inline remap within *Ristra* codes (for example for ALE in several of our reference hydrodynamics codes).

**Part III**

# Deep dives

## 1 *FleCSI*

*FleCSI* is an open-source, compile-time configurable framework designed to support multi-physics application development. As such, *FleCSI* provides a very general set of infrastructure design patterns that can be specialized and extended to suit the needs of a broad variety of solver and data requirements. *FleCSI* currently supports multi-dimensional mesh topology, geometry, and adjacency information, as well as n-dimensional hashed-tree data structures, graph partitioning interfaces, and dependency closures.

*FleCSI* introduces a functional programming model with control, execution, and data abstractions that are consistent both with MPI and with state-of-the-art, task-based runtimes such as Legion and Charm++. The abstraction layer insulates developers from the underlying runtime, while allowing support for multiple runtime systems including conventional models like asynchronous MPI.

The intent is to provide developers with a concrete set of user-friendly programming tools that can be used now, while allowing flexibility in choosing runtime implementations and optimizations that can be applied to future architectures and runtimes.

*FleCSI*'s control and execution models provide formal nomenclature for describing poorly understood concepts such as kernels and tasks. *FleCSI*'s data model provides a low-buy-in approach that makes it an attractive option for many application projects, as developers are not locked into particular layouts or data structure representations.

The structure of applications built on top of the *FleCSI* programming system assumes three basic types of users. Each of the user types has their own set of responsibilities that are designed to separate concerns, and to make sure that development tasks are intuitive and achievable by the associated user type.

The user types are:

- **Core Developer**
  These are users who design, implement, and maintain the core *FleCSI* library. Generally, these users are expert C++ developers who have a well-developed understanding of the the low-level design of the *FleCSI* software architecture. These users are generally computer scientists with expertise in generic programming techniques, data structure design, and optimization.

- **Specialization Developer**
  These are users who adapt the core *FleCSI* data structures and runtime interfaces to create domain-specific interfaces for application developers. These users are required to understand the components of the *FleCSI* interface that can be statically specialized, and must have a solid understanding of the runtime interface. Additionally, specialization developers are assumed to understand the requirements of the application area for which they are designing an interface. These users are generally computational scientists with expertise in one or more numerical methods areas.

- **Application Developer**
  These users are methods developers or physicists who use a particular *FleCSI* specialization layer to develop and maintain application codes. These are the *FleCSI* end-users, who have expertise in designing and implementing numerical methods to solve complicated, multi-physics simulation problems.

The source code implementing a *FleCSI* project will reflect this user structure:

- The project will link to the core *FleCSI* library.
- The project will use one or more specializations (These will usually also be libraries that are linked to by the application.)
- The application developers will use the core and specialization interfaces to write their applications.

For more details on recent developments with *FleCSI*, please see the accompanying slides.

## 2 *Portage*

*A. Herring, C. Ferenbaugh, C. Malone, E. Shevitz, E. Kikinzon, G. Dilts, H. Rakotoarivelo, J. Velechovsky, K. Lipnikov, M. Shashkov, N. Ray, R. Garimella*

*Portage* is an open-source, scalable and extensible remap library for numerical simulations. It supports state-of-the-art remap schemes for meshes and particles in 2D and 3D up to a second-order accuracy. *Portage* ensures critical properties such as local/global conservation and bounds preservation for mesh remap. It enables multi-material field remap through the use of a dedicated interface reconstruction plugin, *Tangram*, and leverages the hybrid parallelism exposed by advanced architectures using multi-processing and multi-threading.

*Portage* is currently the only actively developed open source library that performs locally conservative remap. It provides a lightweight and extensible interface that can easily be customized and integrated into simulation codes. *Portage* supports general polyhedral mesh fields with remap up to a second-order accuracy, while preserving integral quantities of interest and numerical bounds. It supports remap between particle fields as well and provides means to perform mesh remap using the particle remap engine. *Portage* is designed to scale to thousands of cores on distributed architectures through MPI and OpenMP (using Nvidia's Thrust wrapper), with support for GPUs in development.

*Portage* is a framework for creating custom remappers from interoperable components and not a monolithic, catch-all remapping library. It is designed such that its major components can be mixed and matched as necessary as long as they adhere to Portage's API. Its design also seeks to minimize the amount of mesh and field data that must be copied from clients in order to minimize data movement.

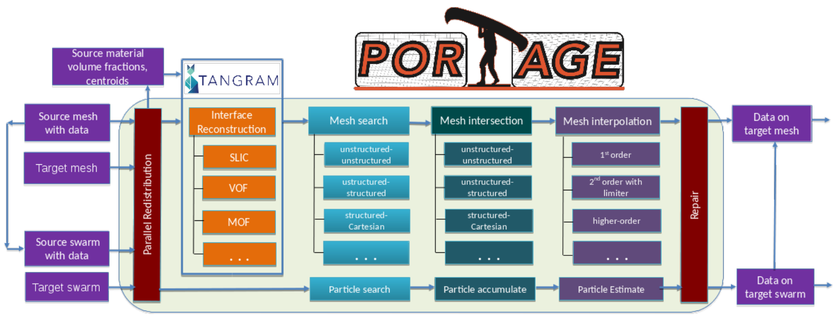


Figure 7: *Portage* software design and workflow

Application developers can:

- Use one of the included drivers with a mix of available and custom components to readily deploy a powerful remapping capability into their application, or

- Write a custom remapping driver and use it with a mix of available and custom components to create a remapping capability uniquely tailored to their application needs.

In order to enable this DIY approach, *Portage* uses a functional design in which the remap driver is templated on the component classes implementing the necessary methods, the mesh and state managers for the source and target. The functional design allows a remap driver to be written such that populating the fields on target entities is a nearly embarrassingly parallel process on-node. Remapping on distributed meshes/swarms is also embarrassingly parallel as long as the target and source partitioning is geometrically matched. On the other hand, if there is a geometric mismatch of the partitioning on the source and target, i.e., source entities overlapping a target entity are on a different node, *Portage* performs some communication and data movement in order to get source mesh cells onto partitions needing them. Once this step is concluded, the remap still shows excellent scaling.

The drivers furnished with *Portage* provide:

- Conservative remapping of multi-material fields between general polygonal/polyhedral meshes
- Higher-order, non-conservative interpolation between particle swarms as well as between meshes and particle swarms
- A modern design templated on the major components - mix and match from the furnished suite or use a custom component
- Direct (no-copy) use of client application's native mesh/particle and field data structures whenever possible (see distributed remap)
- Built-in distributed and on-node parallelism even with custom components (see scaling results)

*Portage* depends on the *Wonton* library to provide mesh/state wrapper interfaces and some common definitions and functionality.

Multi-material remapping requires use of the *Tangram* library for interface reconstruction and optionally, the XMOF2D library. The supplied mesh-mesh remap driver of *Portage* also requires the use of the R3D library for intersection of polyhedra. Distributed parallelism of *Portage* is currently supported through MPI, while on-node parallelism is enabled through the Thrust library. The default on-node parallelism mechanism is OpenMP. Note that the TCMalloc library available in Google Performance Tools is required to see the expected scaling.

### *Implementation and architecture*

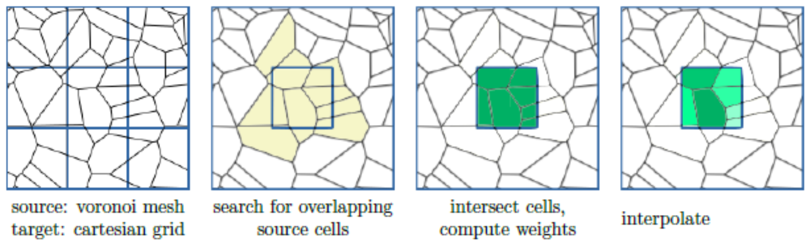

source: voronoi mesh
target: cartesian grid
search for overlapping source cells
intersect cells, compute weights
interpolate

Figure 8: Illustration of intersection-based remap

*Portage* supports three types of remap:

- Intersection-based remap is a conservative scheme that relies on exact intersection of source and target meshes. It first identifies the candidate source cells that may potentially overlap each target cell. It then computes two moments of intersection (volume and centroid) between each target cell and overlapping source cells (Figure 8). Finally, it interpolates the target cell value from the candidate source cells values using the moments of intersection as weights [1].

- Advection-based remap is a conservative scheme specifically designed for meshes with the same topology but with different node positions. As described earlier, this need arises from ALE hydrodynamic simulations when the mesh is slightly smoothed to prevent cell distortion induced by the Lagrangian fluid motion. Here, the remap is formulated as an advection or fluxing of integral quantities in/out of each cell through its faces. Any quantity that is fluxed out of a cell is added into one of its neighbors, so the method is intrinsically conservative. In this algorithm, the interpolation weights are deduced from the flux volumes, which is less expensive but less accurate than the previous remap scheme [17].

- Particle remap is a specific scheme for point clouds. In this method, source fields are reconstructed by means of local regression [8]. Here a shape function is attached to each source point (scatter form) or each target point (gather form). The algorithm first identifies the source points included in the support of the shape function of a target point (Figure 9) which are included in the zone delimited by the user-defined smoothing lengths which control the number of points used for the local regression. It then computes the weights by evaluating the shape function and its derivatives on each point. Finally, it approximates the value on each target point using those weights. Despite its high accuracy, this remap method is not conservative.

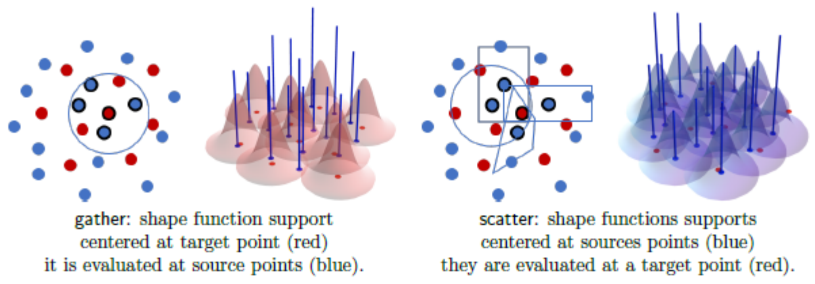

gather: shape function support
centered at target point (red)
it is evaluated at source points (blue).
scatter: shape functions supports
centered at sources points (blue)
they are evaluated at a target point (red).

Figure 9: Illustration of particle-based remap

Each step can be processed in parallel with the granularity of a single point or cell.

## *Design*

*Portage* has a modular design. It relies on extensive C++ templating of all remap steps, allowing client codes to extend, adapt or replace them by customized ones. Most of its core methods are designed to have no side-effects to ease their parallelization and their individual reuse. *Portage*'s components and their interactions are given in Figure 7.

*Portage* takes the source and target domains along with fields data as inputs, and then outputs remapped fields on the target domain. Here a domain can be a mesh or a point cloud. For multi-material fields, it requires the material volume fractions on the source domain as depicted in Figure 10, and which corresponds to the proportion of each material on each cell. The remap workflow consists of six stages:

1. Redistribution: this optional step is only necessary for distributed domains with a mismatch between the source and target partitions. In that case, some source entities (points or cells) are reassigned among MPI ranks such that each target subdomain is overlapped by the corresponding source subdomain. This eliminates the need for communications in the remaining steps.

2. Interface reconstruction: this optional step is only required for multi-material fields and is performed by a dedicated plugin called *Tangram*. It recovers the interface between different materials by computing the material polygons on each source cell given their

volume fractions and, optionally, their centroids for a second-order remap accuracy.

3. Search: this step identifies and retrieves the source entities that are necessary to interpolate the value of a given target entity. The algorithm depends on the remap scheme:

   - *intersection*: collects the source cells that may overlap the target cell.
   - *advection*: collects the source cell itself and a subset of its neighbors.
   - *particle*: collects the source points included in the support of the shape function of a target point in scatter form, and vice-versa for gather form.

4. Computation of weights: this step computes the contribution weights of each identified source entity to reconstruct the value on a given target entity. Again, the algorithm depends on the remap scheme:

   - *intersection*: computes the moments of intersection (volume and cen- troids) of each candidate source cell that overlaps the target cell.
   - *advection*: computes the moments of each swept polyhedron (volume and centroids) formed by the displacement of each face of the source cell.
   - *particle*: computes and accumulates the values of the shape functions and their derivatives on each point given by the search step.

5. Interpolation: this step reconstructs the target entity values by interpolating them using the computed weights. For mesh remap, the gradient of the source field is required to achieve a second-order accurate reconstruction. It is computed in *Portage* by a least-squares method. Here, values can be limited using Barth-Jespersen's limiter, except at domain boundaries because boundary conditions are not yet supported. For particles, we use the term estimation as recovered values may pass *near* the data not necessarily *through* it.

6. Repair: this step is only necessary in case of mismatch between source and target mesh boundaries. Here, remapped values are fixed to enforce the conservation of integral quantities. *Portage* exposes three options to fix partially overlapped cells:

- *constant-preserving* : no field value perturbations but not conservative.
- *locally-conservative*: conservative but perturbations may occur: constant fields may not remain constant.
- *shifted-conservative*: conservative with minimal perturbations but values are shifted: constant field remains constant but with a different value.

It is also possible to extrapolate values to empty cells in the target mesh.

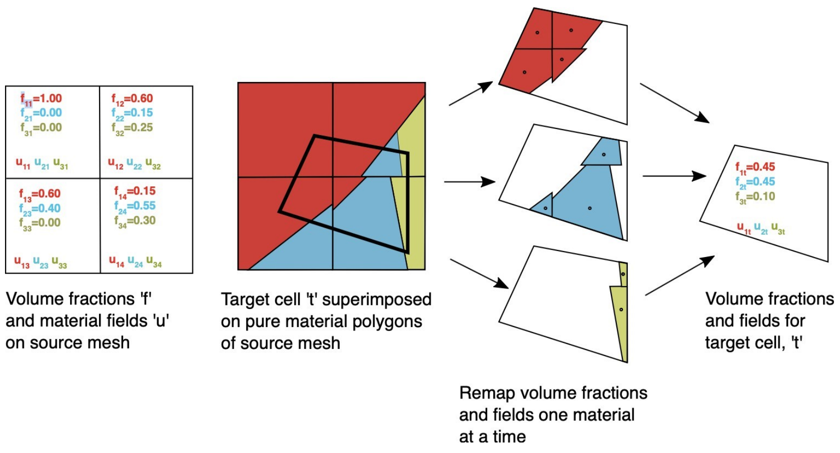

$f_{11}$=1.00
$f_{21}$=0.00
$f_{31}$=0.00
$u_{11}$ $u_{21}$ $u_{31}$
$f_{12}$=0.60
$f_{22}$=0.15
$f_{32}$=0.25
$u_{12}$ $u_{22}$ $u_{32}$
$f_{13}$=0.60
$f_{23}$=0.40
$f_{33}$=0.00
$u_{13}$ $u_{23}$ $u_{33}$
$f_{14}$=0.15
$f_{24}$=0.55
$f_{34}$=0.30
$u_{14}$ $u_{24}$ $u_{34}$
Volume fractions 'f' and material fields 'u' on source mesh
Target cell 't' superimposed on pure material polygons of source mesh
Remap volume fractions and fields one material at a time
$f_{1t}$=0.45
$f_{2t}$=0.45
$f_{3t}$=0.10
$u_{1t}$ $u_{2t}$ $u_{3t}$
Volume fractions and fields for target cell, 't'

Figure 10: Additional step involved in multi-material remap

## *Drivers*

A driver is the interface that exposes the remap capabilities to the client simulation code. Writing a driver allows client codes to mix, match, or extend specialized remap components for their particular needs. *Portage* comes with few drivers to ease the design of custom ones and several apps to show common remap use cases. Each driver is templated on core components (*interface reconstruction, search, weight computation, interpolation*) for each remap method (*intersection, advection, particle*) and on mesh type. If the simulation code provides a mesh with a set of queries that conforms with the mesh wrapper interface, then no data recopy is involved. *Portage* embeds five built-in drivers:

- uberdriver: an easy to use mesh remap class.
- coredriver: a low-level mesh driver that allows finer control on remap steps.

- mmdriver: a legacy monolithic mesh remap driver.
- driver_swarm: a dedicated particle remap driver.
- driver mesh swarm mesh: a mesh remap driver that relies on particle kernels.

## *Scalability*

*Portage* is designed for high performance computing clusters. It relies on both MPI and OpenMP to leverage the hybrid parallelism exposed by such architectures. Here we present some scaling results on a simple multi-material problem in Figure 11. Tests are run on a cluster formed by 256 dual-socket nodes (Intel Broadwell with 18 cores per-socket at 2.1 Ghz). Here we consider a cell-centered three-material field remap with 3D cartesian grids and a simple t-junction material distribution on the domain. The source and target grids have $40^3$ and $120^3$ cells respectively. To ease memory pressure, we set a single MPI rank per node and 16 threads per rank explicitly pinned on cores using KMP AFFINITY=granularity=core,compact.

## *Portage in production*

*Portage* is increasingly being used for inter-code linking for production problems at LANL. Examples can be found in the accompanying slide sets.

Within *Ristra*, *Portage* is of course the main vehicle for intra-code remapping (in ALE for example), but its utility is beginning to pay off elsewhere as the following example makes clear.

### PORTAGE IN EAP

LANL's Eulerian Applications Project (EAP) maintains the production rad-hydro code xRage. xRage contains a remapper capability that maps mesh fields from its native AMR mesh to the GEM mesh (General Eulerian Mesh) format used by some third-party libraries. The current remapper was implemented in a short timeframe and is challenging to maintain.

EAP is currently integrating Portage into xRage as an alternative remapping solution. This required writing some extensions to Portage that were needed by EAP: initial support for cylindrical and spherical coordinates; customized search and intersect routines; and a customized remap driver that efficiently supported both forward and reverse remaps in a single driver. Fortunately, because of Portage's modular design, these customizations could be written easily, and where appropriate, contributed back into the Portage code base.

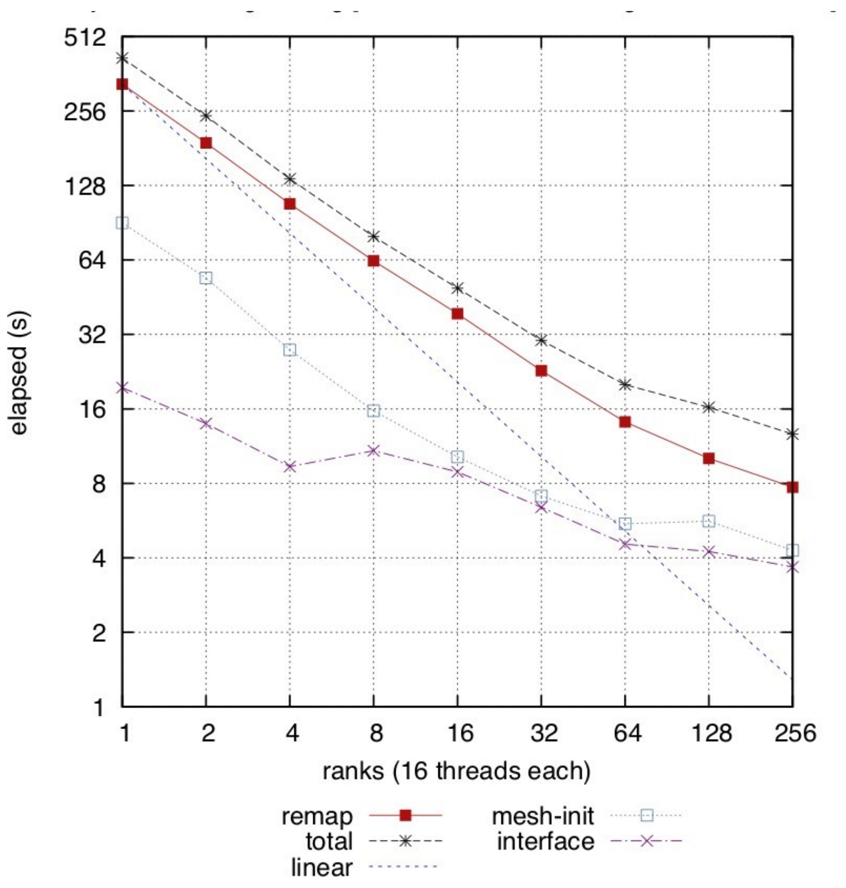


Figure 11: Scaling of multi-material remap in a hybrid parallel setting. The total execution time and the remap time are depicted in black and red respectively. The time spent on material interface reconstruction - which is only performed on multi-material cells - is shown in purple. Here, the workload per rank is impacted by the uneven distribution of multi-material cells. Despite the workload imbalance, a reasonable scaling is still achieved.

A preliminary timing study has shown that the Portage mapper runs faster than the legacy mapper. Two large test cases, typical of expected production use cases, were run.

Case 1:

- AMR mesh: 2.8M cells, distributed
- GEM mesh: 200K cells, distributed
- 800 MPI ranks

Case 2:

- AMR mesh: 2.9M cells, distributed
- GEM mesh: 200K cells, single rank
- 576 MPI ranks

Timing results for these two cases are shown in Table 2. These results come with a few caveats:

- No optimizations have been made for the Portage version yet.
- The mappers are organized differently, so this comparison may not be completely apples-to-apples.

We also analyzed memory usage for Case 2 and found that the legacy mapper used roughly 1.83 Gb, while Portage used roughly 0.24 Gb, a reduction of about 7.5x.

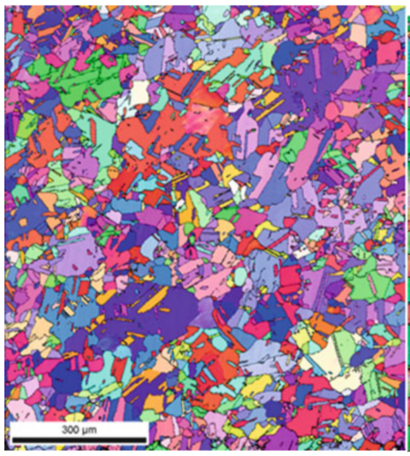
| | Average time per call (ms) | | | | | |
|---|---|---|---|---|---|---|
| | Case 1 | | | Case 2 | | |
| map direction | legacy | Portage | speedup | legacy | Portage | speedup |
| AMR to GEM | 38.5 | 18.4 | 2.09x | 56.4 | 21.5 | 2.62x |
| GEM to AMR | 30.8 | 1.8 | 17.01x | 6754.5 | 302.2 | 22.35x |

Table 2: Preliminary timing results for Portage integration in EAP.

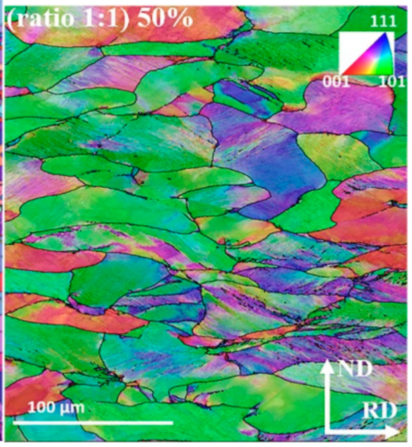
## 3 *Multiscale Material Dynamics on Modern Computer Architectures*

*N. Morgan, J. Moore, R. Lebensohn, M. Zecevic, G. Shipman, V. Chiravalle, D. Holladay, and D. Dunning*

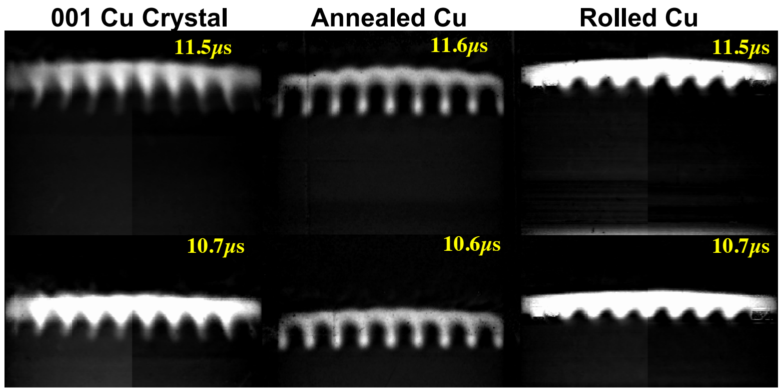
Within the Ristra project, the team developed a novel 3D multiscale hydrodynamic approach to capture mesoscale physics in continuum length-scale simulations. The new approach uses a modern finite element (FE) Lagrangian hydrodynamic method coupled to the multiscale Visco-Plastic Self Consistent Generalized Material Model



(a) Annealed copper [3]



(b) Rolled copper [7]



(c) Flyer plate experimental results [12]

Figure 12: Examples of different kinds of microstructures for metals are shown in (a) and (b). Explosively driven flyer plate experiments demonstrate that the microstructure can greatly affect material dynamics on time scales of $10^{-6}$ seconds, see (c) for comparisons between single crystal, annealed, and rolled copper.

(VPSC-GMM) to capture the nonlinear, non-homogeneous, and directionally dependent behavior of materials within simulations of a full-scale part. The VPSC-GMM is computationally intensive, but is local to each cell so simulations with this model can be significantly accelerated by leveraging modern architectures. This multiscale hydrodynamic approach significantly improves the physical fidelity of material dynamics simulations by allowing complex mesoscale material models to be used on large-scale problems that have materials with high internal variation.

## *Background*

The microstructure of a material varies depending on the composition (alloy composition in metals or crystal-plastic composites in polymer-bonded explosives) and processing (e.g. cast, annealed, rolled, additively manufactured (AM), thermally/mechanically cycled, aged) as shown in Fig. 12. Differences in microstructure can lead to significant differences in the bulk mechanical behavior because of the fundamentally anisotropic, heterogeneous behavior of solids [13], see Fig. 12c. Capturing the mechanics at the scale of the microstructure (termed the mesoscale) requires a constitutive model that represents the mechanics of the individual grains that compose the microstructure. This typically requires mesh resolutions on the scale of tens of microns or smaller while most continuum-scale problems are on the scale of centimeters or larger, making it computationally prohibitive to directly simulate microstructure effects with existing technologies of full-scale parts on exascale machines. As a result, current simulations at the continuum-scale typically use a homogeneous constitutive model to represent the mechanical behavior of the material. The challenges with continuum-scale constitutive models include: (1) homogeneous isotropic models neglect microstructure and material anisotropy; (2) homogeneous anisotropic models must be calibrated for every application and typically neglect microstructure evolution; and (3) all homogeneous models are unable to capture any localized mechanical phenomena such as shear banding, or void nucleation and crack formation/propagation (i.e., damage). An algorithmic gap exists with bridging between mesoscale models and macroscale performance.

Computational physics is facing additional, new challenges that arise with modern computer architectures comprised of multi-core CPUs combined with GPUs (termed heterogeneous), but these architectures also create new opportunities to support higher-fidelity physics. Heterogeneous computer architectures are forcing the simulation community to reconsider long held paradigms, such as relying on mesh refinement with lower-order methods, and are shifting the algorithmic development towards higher compute intensity methods with excellent data locality and internal parallelism. The goal is to maximize the compute intensity per memory load (FLOP/memory ratio) and to eliminate the communication between distance cells (e.g., good data locality). A new multiscale hydrodynamic simulation approach was developed in the Ristra project that is: 1) well suited for homogeneous and heterogeneous computer architectures; 2) is able to predict the 3D mechanical behavior of materials across different compositions and processes, and that is sensitive to microstructure heterogeneity; and 3) designed for shock driven material dynamics.

### *Innovation*

As part of a multiscale strategy, different homogenization models can be used to connect single crystal and polycrystal behaviors. This work is focused on the use of self-consistent (SC) homogenization, in which single crystals deform differently according to their orientation and strength. SC methods provide more accurate microstructure-sensitive constitutive response of the polycrystalline material points. In this work, we couple the multiscale Visco-Plastic Self Consistent Generalized Material Model (VPSC-GMM) with a modern finite element (FE) Lagrangian hydrodynamic method [5] in each cell of the mesh, see Fig. 13. With this approach, the meso and macro-scales are separated, but it is more predictive and accurate than using anisotropic material models that are calibrated a priori to a particular test case. The new approach is computationally expensive compared to using anisotropic strength models, but it can account for microstructural evolution and how the latter affects the macroscopic behavior, especially under deformation conditions involving large deformations and/or large dynamic compression and strain-rates. The high numerical cost associated with using VPSC-GMM in every cell of the mesh in a simulation is dealt with and overcome by using novel methods, efficient software implementation, and leveraging parallelization strategies for modern computer architectures.

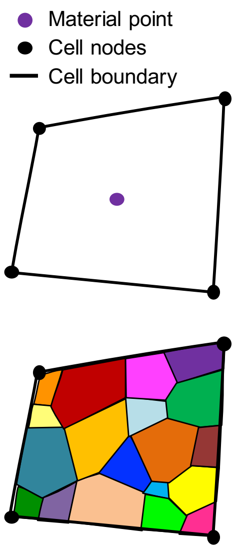


Figure 13: The VPSC-GMM seeks to capture the dynamics of individual crystals (also called grains) within a cell and will use a self-consistent homogenization method to create a cell average stress, which is stored at the material point. Each crystal can have a unique orientation and strength model. The stress at the material point is used in the FE Lagrangian hydrodynamic code [5] to evolve the continuum-level fields in the cell forward in time. The strain of the cell is then used by the VPSC-GMM to evolve the stress, strain, and orientation of each crystal, and then it will return a cell average stress. The process is repeated every time step in the hydrodynamic simulation.

## *New numerical methods in VPSC-GMM*

In this section, two main improvements to the approach described in the previous section, which are necessary for simulating more complex problems using VPSC-GMM within an explicit FE solver, are presented. First, we developed a more robust implicit solver to evolve the crystal orientations and find the stress and strain of each crystal. Next, a linear extrapolation algorithm for fast calculation of mesoscopic stress is presented. We end this section providing further details of the VPSC-GMM numerical implementation.

## LINEAR EXTRAPOLATION OF MESOSCOPIC STRESS

Solving the full elasto-viscoplastic SC problem for each polycrystalline material point at each increment is computationally very expensive and impractical. To get around this, the elasto-viscoplastic SC solution is not calculated at every increment, but only after enough strain has accumulated with respect to the previous increment at which the full SC problem was solved. In the increments between the full solutions, the mesoscopic stress is approximated using a linear extrapolation method.

In this work, we developed a new extrapolation procedure which accurately takes into account changes in the loading direction. The main idea behind this new extrapolation procedure is based on the assumption that, for small strain increments, the non-linear dependence of viscoplastic strain rate on mesoscopic stress can be accurately described by the linear effective relation given by

$$\dot{\epsilon}^{v} = \bar{M}^{v} : \sigma' + \bar{\dot{\epsilon}}^{v0} \tag{1}$$

Where $\dot{\epsilon}^{v}$ and $\sigma'$ are mesoscopic viscoplastic strain rate and deviatoric stress, and $\bar{M}^{v}$ and $\bar{\dot{\epsilon}}^{v0}$ are effective viscoplastic compliance and back-extrapolated strain rate. The integrated strain rate then becomes linear in mesoscopic stress and can be easily solved. In our approach, the full elasto-viscoplastic polycrystalline problem is solved every $l$ increments, after enough von Mises strain has accumulated to cause appreciable evolution in state variables and thus mesoscopic response (see fig. 14). The set of $l$ increments applied at a material point is replaced by one accumulated increment, which is then applied to the elasto-viscoplastic polycrystal. The strain increments are accumulated as:

$$\Delta\epsilon^{acc,n} = \sum_{i=n-l+1}^{n} \Delta\epsilon^{i,rot} \tag{2}$$

where $\Delta\epsilon^{i,rot}$ are strain increments rotated to the configuration at current increment $n$. The rotation increments are used to update the total amount of rotation per grain for that step. The threshold von Mises strain increment, $\Delta\epsilon_{vm}^{th} \in [0.0001, 0.01]$, is automatically updated based on the difference between the stress calculated by solving the full elasto-viscoplastic problem and the stress calculated by linear extrapolation, which is described next. In the increments $k \in (n-1, n)$ between the full solutions of the elasto-viscoplastic problem, the stress is calculated by linear extrapolation. The integrated strain rate is solved for stress under assumption that viscoplastic effective properties are independent of the mesoscopic stress

(linear approximation of viscoplastic behavior).
The calculated stress is then rotated to global frame and returned to Lagrangian FE hydrodynamics code. The elastic self-consistent effective stiffness at increment $k$ is approximated by last available estimate, $\bar{L}^{e,m,k} = \bar{L}^{e,m,n-1}$. The viscoplastic effective compliance and back-extrapolated strain rate at $k$ are approximated using the forward euler method, and the time derivatives of the effective viscoplastic compliance and back-extrapolated strain rate are calculated using a finite difference approximation. The available information on the dependence of viscoplastic strain rate on stress is utilized so that the change in the direction of $\Delta\epsilon^k$ directly affects the direction of stress through both the elastic and viscoplastic strain rates.

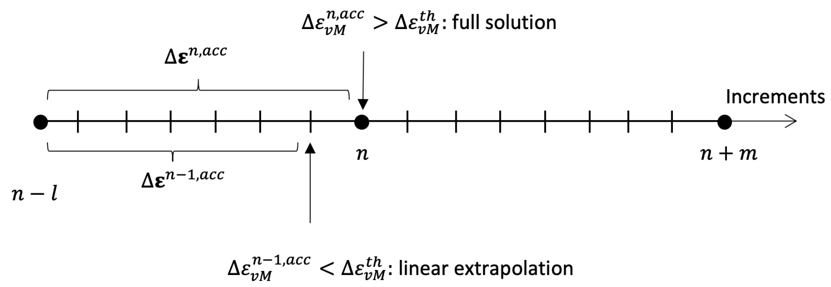

$\Delta\varepsilon_{vM}^{n,acc} > \Delta\varepsilon_{vM}^{th}$: full solution
$\Delta\boldsymbol{\varepsilon}^{n,acc}$
Increments
$n$
$n+m$
$n-l$
$\Delta\boldsymbol{\varepsilon}^{n-1,acc}$
$\Delta\varepsilon_{vM}^{n-1,acc} < \Delta\varepsilon_{vM}^{th}$: linear extrapolation

Figure 14: Based on the magnitude of the accumulated strain increment at current increment, the algorithm decides whether to solve the full elasto-viscoplastic problem or to perform linear extrapolation. The extrapolation is uesd as long as the accumulated strain at increment $n$, $\Delta\epsilon^{n,acc}$ is below some threshold value, $\Delta\epsilon_{vm}^{n,th}$. This method accounts for the possible change in the direction of loading, thereby increasing the accuracy of the extrapolation.

## *Verification and Validation*

The Fortran-C++ interface allows coupling of the new VPSC-GMM model with the fully-conservative Lagrangian FE hydrodynamics code. In section *Verification: single-element uniaxial quasistatic tension, and compression and simple shear of an fcc polycrystal*, we validate the VPSC-GMM coupling and algorithmic modifications by simulating uniaxial quasistatic tension, compression, and simple shear of an fcc polycrystal using a single element and comparing the texture evolution to the stand-alone VPSC code. Section *Single-element hydrodynamic compression of an fcc polycrystal* compares the hydrodynamic response of a single element undergoing high-rate compression using VPSC-GMM coupled to the FE hydrodynamic code and using VPSC coupled to the commercial code Abaqus$^{TM}$. Section *Validation on a polycrystalline Taylor anvil experiment* compares the results from a simulated dynamic Taylor anvil experiment to the experimental results of a textured Ta cylinder, which is commonly used to validate material models at high strain rates. Section *Validation on a single-crystal Taylor anvil experiment* presents results for a series of single crystal Taylor anvil experiments.

## VERIFICATION: SINGLE-ELEMENT UNIAXIAL QUASISTATIC TENSION, AND COMPRESSION AND SIMPLE SHEAR OF AN FCC POLYCRYSTAL

The VPSC-GMM with the FE hydrodynamics code was run for a single element representing an fcc polycrystal with an initial texture of 100 grains with random orientations, for uniaxial tension and compression and simple shear, applying a strain rate of 1 $s^{-1}$ with no hardening (Fig. 15). The fcc single crystal grains were assumed to deform plastically by $\{111\}\langle 1\bar{1}0\rangle$ slip, with a constant (i.e. no strain-hardening) slip resistance $\tau_c^{s(r)}$ =1.0 GPa, a rate exponent n=20, and initially spherical shape. The elastic constants correspond to austenitic steel: $C_{11}$= 205.0 GPa, $C_{12}$=138 GPa, and $C_{44}$=126 GPa. The simulations were run up to 50% total strain for tension, compression, and shear. The results are compared to those obtained with the stand-alone version of VPSC.

The resulting textures for each test case are compared in Fig. 16. The red points are the stereographic projections of (111) poles of the 100 grains predicted by the stand-alone VPSC code and the blue points correspond to the predictions from the FE hydrodynamics code with VPSC-GMM. The results match almost exactly for all cases, with the largest difference observed in the shear case, due to slight difference between how the FE hydrodynamics code and the stand-alone VPSC code handle macroscopic rotations. Also, the stand-alone VPSC code assumes the material is incompressible, while the FE hydrodynamics code allows for compressibility. To make the hydrodynamics code match as well as possible to the incompressible solution, the individual nodes of the element were given a velocity to match the required deformation. Fig. 17 shows the stress strain response for the shear case. Since hardening was not active for this calculation, the slight positive slope of the plastic region comes from geometric hardening caused by texture evolution. Also, the sampling rate of the calculation was too coarse to show the elastoplastic transition of the material. The first sample is at zero strain and the next sample jumps to plastic deformation. This was done because the stand-alone VPSC code does not include elasticity, so a comparison has little meaning.

For uniaxial tension and compression, the stand-alone version of VPSC takes in a strain tensor whose longitudinal component is defined while the transversal components are calculated with an implicit solver to enforce incompressibility. Slight variations between the implicit solver in the stand-alone VPSC code and the solution from the hydrodynamics code created slight differences in the input velocity gradient.

## SINGLE-ELEMENT HYDRODYNAMIC COMPRESSION OF AN FCC POLYCRYSTAL

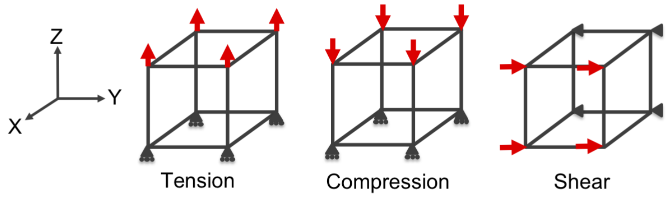
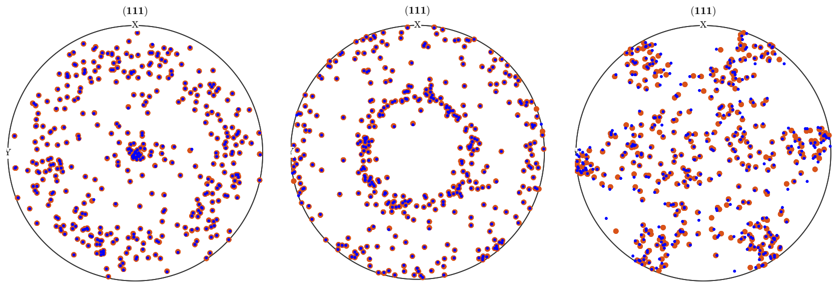


Figure 15: Schematics of the three verification test cases.



(a) Tension (b) Compression (c) Shear

Figure 16: The texture predictions using the VPSC-GMM coupled to the FE hydrodynamics code are compared to the stand-alone version of VPSC for an fcc polycrystal with initial random texture. The red dots are the stand-alone VPSC predictions and the blue dots represent the texture predicted using the FE hydrodynamics code with VPSC-GMM. The results are given for a single element undergoing tension, compression and simple shear.

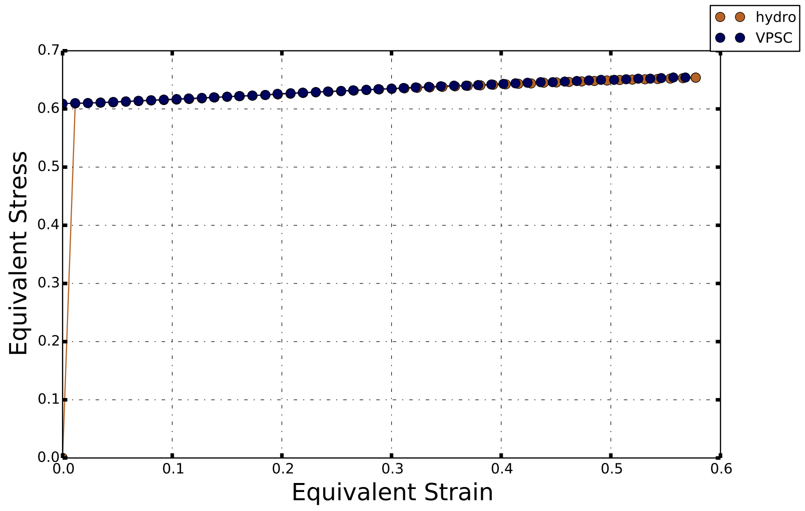


Figure 17: Effective stress-strain response for the shear test case predicted with the stand-alone VPSC code (blue dots) and the VPSC-GMM coupled to the hydrodynamics code (orange dots). The stress values are in MBars. The stand-alone calculation neglects the material's elastic response, which is why the starting stress value is non-zero, while for the hydrodynamics code the strain-stress curve starts at zero stress. The small positive slope in the plastic region is caused by geometric hardening, since no strain hardening was used for this calculation.

The FE hydrodynamics code with VPSC-GMM was also compared to the VPSC model coupled to Abaqus$^{TM}$ Explicit for a single-element high rate compression case. This test case is exploring the hydrodynamic regime, where the previous tests were all quasistatic. Symmetry boundary conditions were used for this test case on the x=0, y=0, and z=0 planes, and an initial velocity of 4000 m/s was applied downward to the nodes on the top face of the element. This simulation was run for one microsecond. The purpose of this test was to compare only the high strain rate hydrodynamics behavior of the coupling, not necessarily the accuracy of the physical behavior. The material input files used for this test were the same as the test done in section *Verification: single-element uniaxial quasistatic tension, and compression and simple shear of an fcc polycrystal*. Plots of the density, pressure, volume, and internal energy from Abaqus$^{TM}$ and the FE hydrodynamics codes are given in Fig. 18. There is excellent agreement between the predictions of both codes.

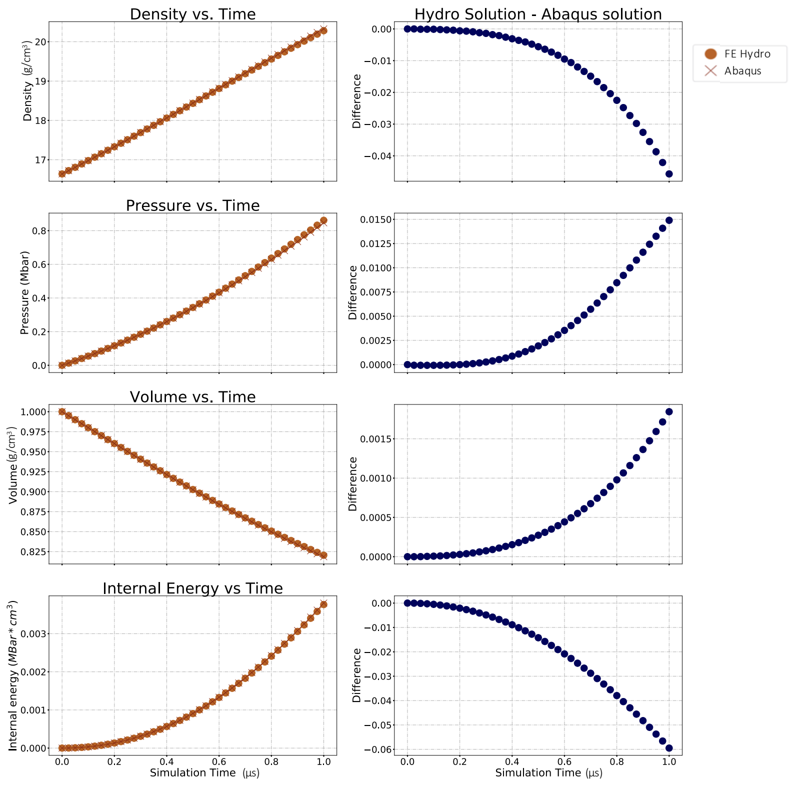


Figure 18: Results from Abaqus$^{TM}$ are compared to results from the FE hydrodynamics code with VPSC-GMM for a unit cell undergoing rapid uniaxial compression for one microsecond. In the left column, the orange dots represent the solution from the FE hydrodynamics code and the red crosses are the solution from Abaqus$^{TM}$. The blue dots in the right column are the difference in the solutions.

## VALIDATION ON A POLYCRYSTALLINE TAYLOR ANVIL EXPERIMENT

The Taylor impact experiment involves impacting a cylindrical specimen against a stiff target at high rate (∼100 m/s), and measuring the deformed shape and microstructure after the impact. We simulated Taylor impact experiments of textured Ta cylinder performed by Maudlin et al. [11]. Ta cylinders were cut from a rolled Ta plate, with the cylinder axis parallel to the rolling and transverse directions. Cylinder diameter was 7.62 mm and cylinder length was 38.1 mm. The impact velocity was 175 m/s.

Due to symmetry of the specimen and the constitutive response, one quarter of the cylinder was simulated and discretized into 3597 cells. The nodes in the planes of symmetry were allowed to move in those planes only. The contact between the anvil and the cylinder is simulated by constraining the displacements of cylinder nodes in the plane of contact to that plane. An initial velocity of 175 m/s was imposed to the cylinder. The density of Ta cylinder is 16640 kg/m$^3$ [11]. The polycrystalline Ta's initial microstructure was the same for each element, and consisted of 419 equiaxed grains, with orientations chosen to reproduce the measured initial texture of the rolled Ta plate. The initial slip resistance $\tau_0^s$=115 MPa and rate exponent n=14 for Ta were calibrated based on measured through-thickness compression initial yield stresses of Ta specimens cut from the same rolled plate as cylinders and measured by Chen et al. [4] (Fig. 19). For simplicity, no strain-hardening was assumed and grains were allowed to deform by $\{1\bar{1}0\}\langle 111\rangle$ and $\{11\bar{2}\}\langle 111\rangle$ slip.

Fig. 20 shows the mesh, and the cylinder shape and von Mises stress at different time increments. In the initial stages of the simulation, the von Mises stress is the highest at the foot of the cylinder where deformation is the highest (Fig. 20b). The tail of the cylinder remains below the yield point throughout the simulation. At later stages, the peak von Mises stress moves away from the foot more towards the cylinder center (Fig. 20c-e). Note that von Mises stress is proportional to the strain rate, and the shift of peak stress from foot toward the tail indicates propagation of the plastic wave.

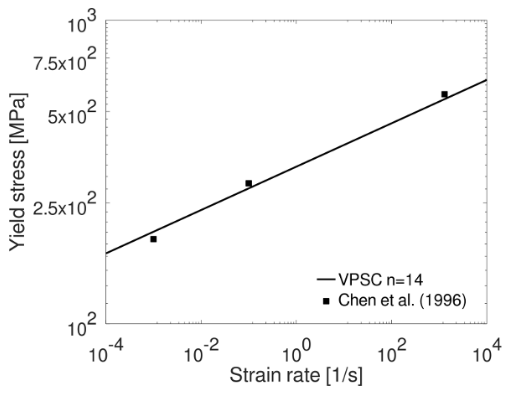


Figure 19: Rate sensitivity of initial yield stress under through thickness compression predicted by VPSC, compared to measured initial yield stresses at strain rates of 0.001, 0.1 and 1300/s, which were extracted from experimental compressive stress strain curves (Chen et al. [4] ).

Fig. 21 compares the predicted and measured deformed cylinder

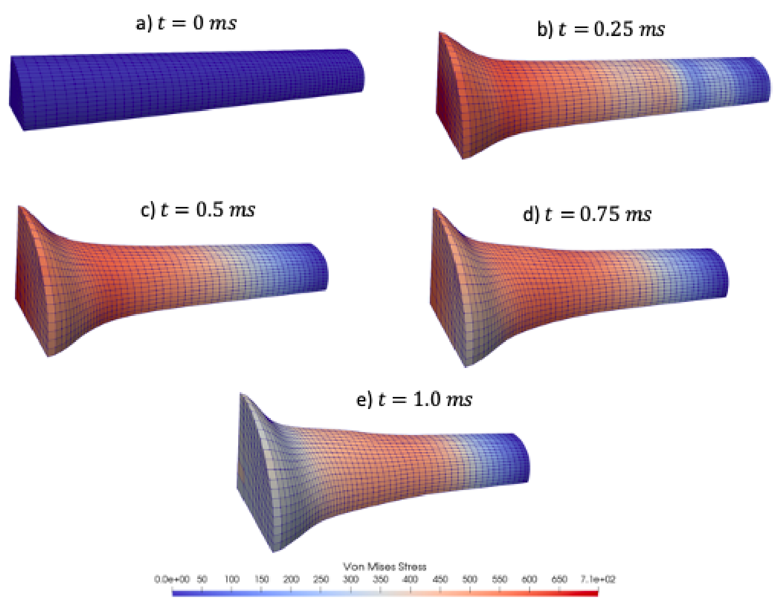


Figure 20: The evolution of von Mises stress and shape of Ta cylinder during Taylor impact simulation at five different times: (a) 0 ms, (b) 0.25 ms, (c) 0.5 ms, (d) 0.75 ms, (e) 1.0 ms.

shapes. Fig. 21a shows good agreement between the predicted and measured shapes of the cylinder foot. Comparison of the deformed shape to the initial cylinder shape at the foot (dashed circle) indicates a very high degree of deformation and significant ovalization. During rolling of the initial Ta plate, the $\{111\}$ crystallographic planes/directions tend to align with the normal direction (through thickness direction of the plate). The anisotropic single crystalline response of bcc Ta is hard along this direction, meaning that the through thickness direction of the rolled plate is a hard direction in comparison to in-plane directions. Since the hard direction is parallel to the x-direction of the cylinder, the cylinder deforms less in the x-direction than in the y-direction resulting in the observed anisotropic shape. Major and minor profiles and radial strains shown on Fig. 21b and Fig. 21c, respectively, reveal even larger ovalization away from the foot. As the deformation progresses, the crystallographic orientations reorient and the anisotropy of polycrystalline constitutive response changes. Since the ovalization is not the strongest at the foot where most of the strain is accumulated, it appears that difference between the x- and y-cylinder dimensions reduces with straining, due to texture evolution.

The boundary conditions implemented in the FE hydrodynamics code simulation do not allow for the cylinder to "bounce back" after the initial contact. This difference in the experimental vs simulation setup contributes to the discrepancy between the calculated and the experimental results. In addition, due to very high strain rates and strains, especially at the foot of the cylinder, the temperature increase due to adiabatic heating is significant (according to simulations of Zecevic and Knezevic [18] temperature at center of the foot reaches values close to 1000°C). The temperature increase would lead to a decrease of the yield stress, making the foot softer at higher strains. This effect is not captured in our model, and the predicted deformation at the foot is thus lower than experimentally observed.

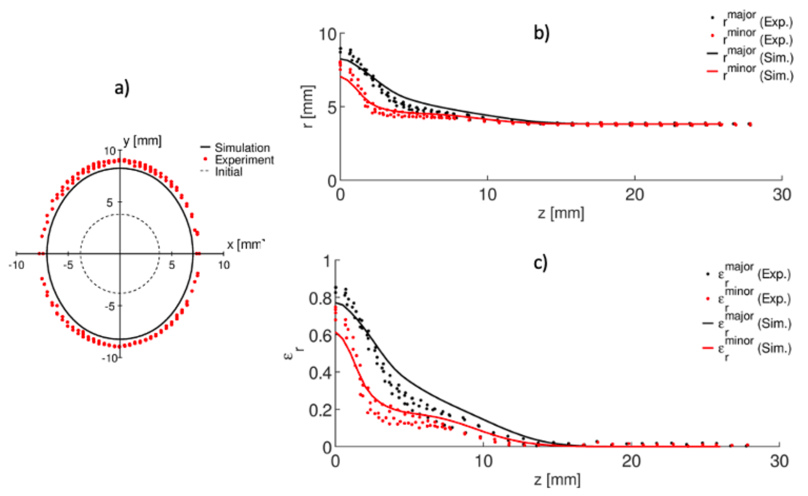


Figure 21: Comparison between simulation (solid line) and the experimental results (dots): (a) foot of the Taylor cylinder after the impact (before impact shown with a dotted line); (b) major and minor radius of the cylinder after the impact; (c) major and minor radial strain after deformation.

## VALIDATION ON A SINGLE-CRYSTAL TAYLOR ANVIL EXPERIMENT

The VPSC-GMM coupled to the Abaqus$^{TM}$ explicit code was used to simulate single crystal (SX) Ta Taylor Anivil tests by Lim et. al. [9]. The results are shown in Fig. 22. Credit for the SX work goes to Miroslav Zecevic (T-3), Jesse Feng (Univ. NH), Marko Knezevic (Univ. NH) and Ricardo Lebensohn (T-3). The VPSC-GMM captures the anisotropic deformation in these experimental tests, which demonstrates the predictive capabilties of this multiscale model.

### *Performance improvements using openMP*

The VPSC-GMM model involves significantly more computational work than the FE hydrodynamics solve. The extrapolation scheme added to VPSC-GMM increases the speed of a simulation, but it can create load imbalance issues. Figure 23 shows how the computational work is not always evenly distributed over the CPU cores with a notional Taylor anvil simulation. Each color in Fig. 23 is a CPU core running a single thread that executes the computational work for a set of elements in the mesh. In this notional example, the orange core is doing most of the work in the calculation, because VPSC-GMM is executed on the orange core while the other cores will execute the less expensive extrapolation scheme. This means that the code is

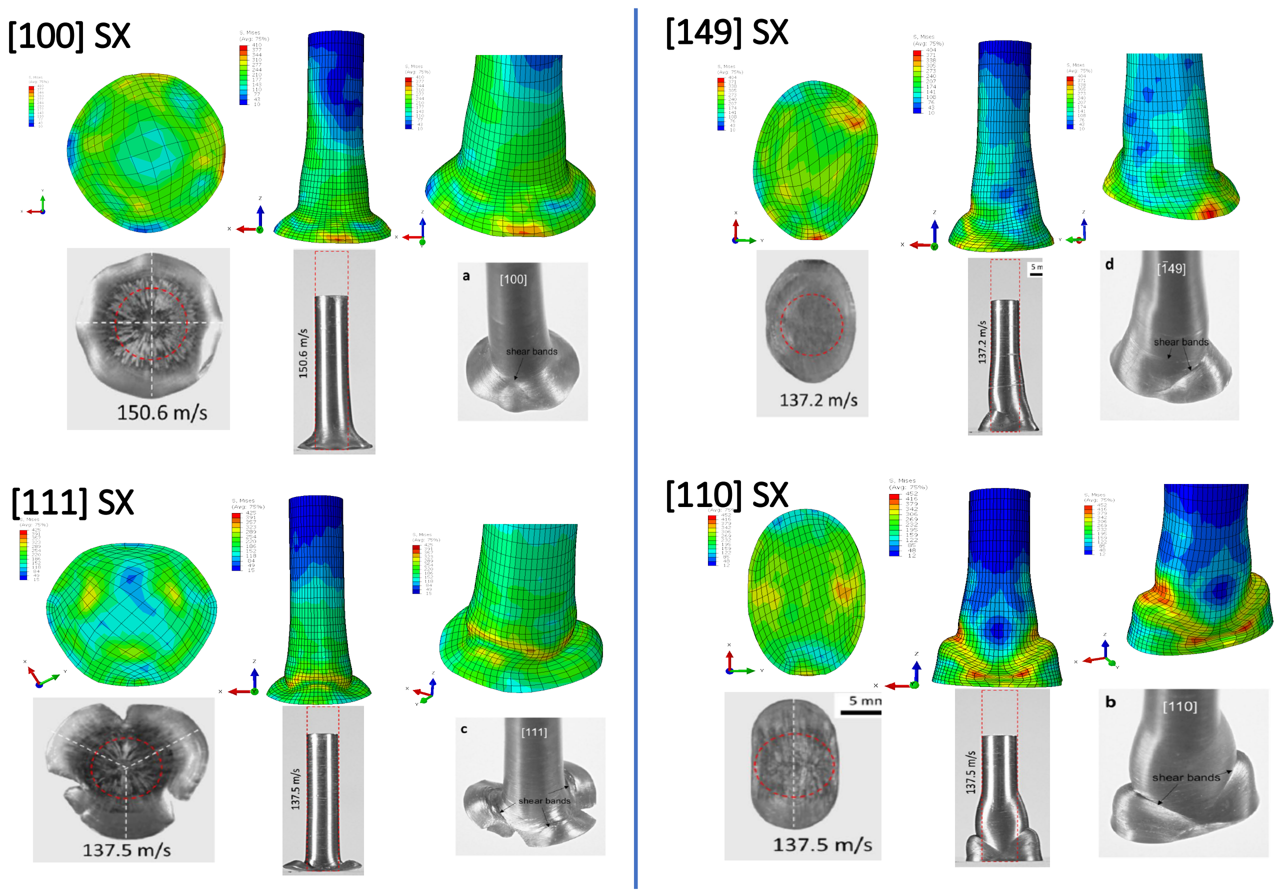


Figure 22: The VPSC-GMM, can be embedded in different solid mechanics/dynamics solvers. Abaqus$^{TM}$ VPSC-GMM results are shown for the single crystal (SX) Ta Taylor-Anvil experiments by Lim et al. [9]. The results match the experiments remarkably well.

effectively running in serial, which is sub-optimal.

One way to get around the load imbalance issue is to change the stride pattern for walking over the elements within a loop to help prevent neighboring elements from running on the same CPU core. This is achieved by creating a stride array so that a loop will jump through the initial mesh to help distribute the work over the cores. Figure 24 shows a notional example of this process with an 8 core CPU and a stride size of 8. The efficacy of this improvement is problem dependent. If all elements in the mesh are undergoing large amounts of deformation then the load is reasonably balanced over the CPU cores. That being said, for problems were only small regions of the mesh are experiencing significant deformation, then using a stride pattern to walk over the mesh greatly accelerates the calculation (i.e., it decreases the runtime) and improves the scaling performance. It should be noted that the stride size is a user settable variable, and the optimal stride size for each simulation will be different. In our studies using multi-core CPUs, we found a stride size equal to the number of threads used in the simulation will, on average, distribute the work evenly across the CPU cores. All scaling results presented in this paper use a stride size equal to the number of threads.

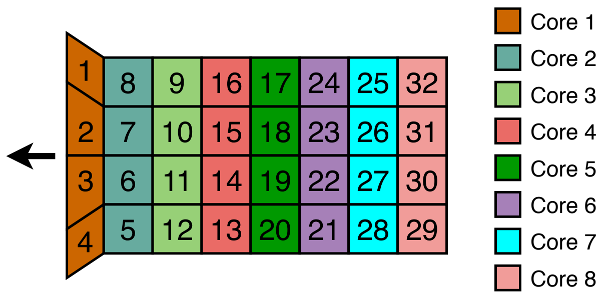


Figure 23: The load imbalance issue is shown for a notional Taylor anvil simulation using an 8 core CPU with 8 threads (i.e., 1 thread per core). Each color represents a CPU core and the numbers correspond to the element index. Depending on how the mesh is walked over in a loop, it is possible (and likely) that the VPSC-GMM will be executed on the same core or just a few cores; as a result, majority of the computational work is on a core or just a few cores. This load imbalance issue is addressed in this work.

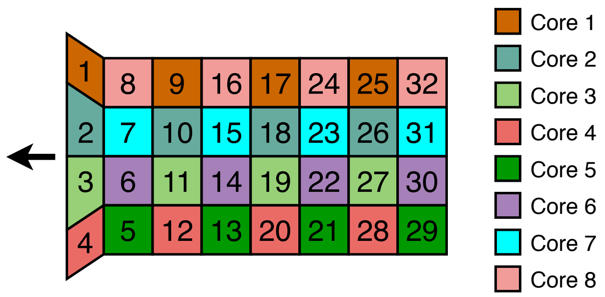


Figure 24: The amount of computational work in each element can vary substantially across the mesh. In this notional example, an 8 core CPU is used with 1 thread per core. The elements in the loop are walked over using a stride size of 8, which better balances the work load over the 8 CPU cores.

## OPENMP PERFORMANCE STUDIES

To test the runtime performance of the new multiscale approach, the Taylor impact simulation (see section *Validation on a polycrystalline*

| Architecture | AMD | ARM | KNL | Skylake |
|---|---|---|---|---|
| CPU | AMD-EPYC 7551 | Cavium ThunderX2 CN9975 | Xeon Phi 7250 | Xeon Platinum 8176 |
| cores | 32 | 32 | 68 | 28 |
| Memory bandwidth [GB/s] | 170.6 | 158.95 | 115.2 | 119.21 |
| Cache memory [MB] | 64 (L3) | 8 (L2) | 34 (L2) | 28 (L2) |

Table 3: The performance of the VPSC code was tested on a series of multi-core CPUs. The details on each CPU are provided.

*Taylor anvil experiment*) was run for 50 equally spaced time steps while increasing the number of threads up to the maximum number of cores, and then with hyperthreading. The first 50 times steps are very computationally costly and there is a large variation in the computational work load across the mesh because the elements in contact with the wall are undergoing very high strain rates relative to the rest of the mesh. The goal is show linear scaling. This scaling analysis was done on four different CPU architectures (AMD, ARM, KNL, and Skylake) to ensure the performance increase was not machine-dependent (CPU specifications are given in Table 3). Parallelization was done using Fortran OpenMP threads over the VPSC-GMM code. The OpenMP scaling was aided by vectorization in the VPSC-GMM code.

The results of the scaling analysis are shown in Fig. 25. The scaling is linear on all architectures up to the maximum number of cores, as desired. When pushed passed the core limit into hyperthreading, different architectures showed more variance. In the hyperthreading regime, the KNL and ARM architectures showed a very slight performance increase with more threads, considerably weaker when compared with linear scaling up to that point. The AMD and Skylake CPUs showed performance decrease with hyperthreading. This is quite common, when the task of scheduling all these threads takes more of a performance toll than the additional work that can be done. AMD, ARM, and Skylake runs performed similarly when using the number of threads equal to the max number of cores on the CPU.

The vectorization-specific data was gathered on an Intel Skylake processor. This CPU was chosen simply because it is the architecture that we had the best tools available for gathering and analyzing such data. We used Likwid [16], a performance monitoring application that provides many architecture-specific hardware counters. The result from two separate runs are shown in Table 4 corresponding to the case of using vectorization and with vectorization turned off. We see that vectorization allows for a large increase in the number of double-precision FLOPS per second. This improves the flat runtime performance of the code, and also greatly increases the memory bandwidth that is achieved throughout the calculation. Additionally, we see a sharp decrease in the amount of energy used by the VPSC

| Measure | Vectorized | Non-vectorized |
|---|---|---|
| Runtime (s) | 14.9287 | 70.8878 |
| Energy consumed (J) | 37.0646 | 120.8417 |
| Memory Bandwidth (MBytes) | 9.9381 | 0.9814 |
| Vector FLOPS/s | 115.0494 | 0.0016 |
| Arithmetic Intensity | 0.3722 | 0.2128 |

Table 4: The VPSC code had favorable performance using vectorization. All metrics notably improve compared to the calculation without vectorization.

code. The arithmetic intensity is the ratio of floating point operations (FLOPs) to bytes needed from memory; a higher value means that more FLOPs per memory load, which is desirable. The arithmetic intensity increases using vectorization.

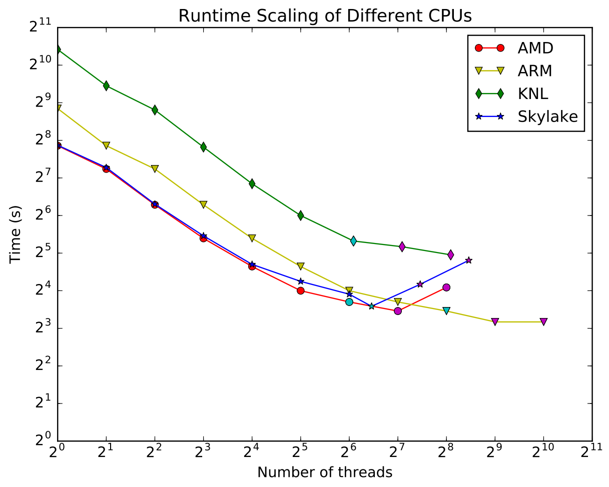


Figure 25: The runtime scaling study for four different CPU architectures: AMD, ARM, KNL, and Skylake. The cyan filled shapes represent the calculations where the number of threads is equal to the max number of cores on the CPU. The purple filled shapes show results for hyperthreading (i.e., using more threads than the number of cores on the CPU). The scaling is nearly linear up until the maximum number of cores on all architectures. Performance gains were seen with hyperthreading for the ARM and KNL CPUs, while for Skylake and AMD the hyperthreading slowed down the code.

## *Performance studies with LEGION*

Multiscale material dynamics presents unique challenges in achieving performance and scalability on modern heterogeneous architectures. As an example, the FE hydrodynamics code has much different strong and weak-scaling characteristics relative to the VPSC-GMM code. Additionally, these two codes have different performance characteristics when using CPU or GPU resources that can be problem and scale dependent. Historically the challenge of optimally mapping different parts of these codes to different compute resources would reside with the application developer often requiring heroic efforts in coding and performance analysis and tuning.

The Legion programming system [2] addresses these concerns by providing high-level abstractions for application and middleware developers to describe properties of the data on which they operate, enabling the underlying runtime to automate many aspects of achieving high performance, such as extracting task- and data-level parallelism. Legion is designed to automate details of scheduling tasks and data movement (performance optimization), and separates the specification of tasks and data from the mapping onto a machine (performance portability). Legion inter-operates with other runtime systems such as MPI, allowing incremental adoption in existing applications.

Supercomputers such as the Sierra supercomputer at LLNL provide the majority of compute performance in terms of flops and memory bandwidth via GPUs. The Sierra node architecture consists of dual socket IBM Power9 CPUs each with 22 cores coupled to 4 NVIDIA Volta GPUs interconnected via NVLink technology. VPSC was ported to the Volta GPUs using CUDA Fortran to minimize the required changes to the code base. VPSC was then refactored as a stand-alone library that built upon Legion for inter-node (and multi-GPU) parallelization. This combination enables the VPSC library to handle all the complexity of data movement across different memory spaces and scheduling of VPSC tasks across the machine. These complexities are hidden from the user of the library but can be controlled through a mapping interface within Legion to specify application and machine specific policies to achieve high performance and scalability. This decomposition is illustrated in Fig. 26, wherein the FE code can use MPI for parallelization across Sierra's Power9 CPUs while Legion controls the distribution of VPSC tasks across the Volta GPUs.

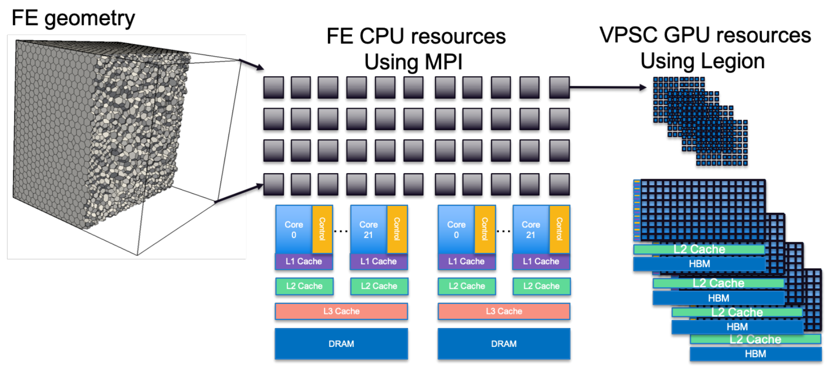


Figure 26: Decomposition of the FE+VPSC codes using MPI and Legion runtimes on Sierra.

Performance of VPSC was first assessed on GPU vs CPU on the Sierra node architecture (RZAnsel). As detailed in Fig. 27, overall performance of the GPU exceeded that of the CPU as the number of cells per GPU exceeded 60.

Next the scalability of Legion VPSC was assessed on the Sierra node architecture (RZAnsel) up to 16 nodes and 64 GPUs. As detailed in Fig. 28, we achieve nearly perfect weak scaling with Legion scheduling VPSC GPU tasks entirely within a standalone library.

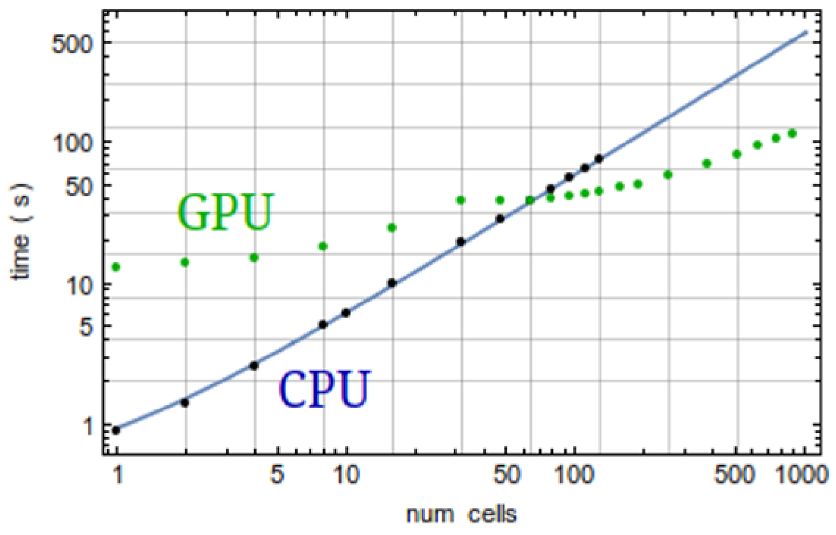


Figure 27: Time to solution for GPU vs CPU VPSC tasks on Sierra node architecture.

Consumers of this library (such as FE) are insulated from this complexity. Ongoing work is focused on integration of the Legion VPSC library into the FE hydrodynamic code base to demonstrate fully coupled simulations at scale.

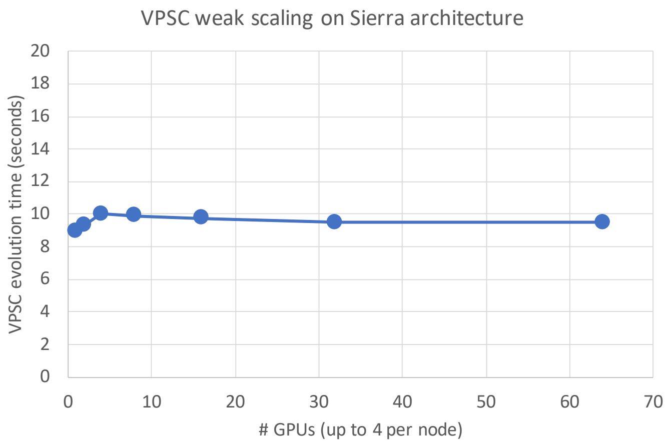


Figure 28: Weak scaling of Legion VPSC on Sierra node architecture up to 64 GPUs.

## *Conclusion*

LANL requires tools to quantify the performance of materials made with new and existing manufacturing processes. To address this need, the VPSC-GMM was created and coupled to a finite element (FE) Lagrangian hydrodynamic method. A series of verification and validation tests were performed. The results from the test show the approach can accuracy simulate the experiments.

This research fills an existing technology gap by delivering a high-fidelity multiscale hydrodynamic approach to practically predict the performance of composite, cast, wrought, aged, and AM materials. The ability to predict the mechanical behavior of complex materials in dynamic environments is beneficial to Laboratory missions in a variety of ways. Predictive modeling of materials is useful for (1) contrasting the influence of microstructure texture on material behavior such as quantifying the performance differences between cast and rolled metals, (2) determining conditions that lead to material damage and failure, (3) predicting the influence of advanced manufacturing process such as additive manufacturing on material properties, and (4) aiding in the design of various materials like metals or high explosives. Furthermore, the new multiscale approach will be useful for general materials research and could lead to greater opportunities for cooperative work in academia.

**Part IV**

# Productivity in the *Ristra* Environment

## 1 *The physics developer experience*

The development of a practical multiphysics application necessitates flexibility. Users must be able to select which physics models to include in their simulations at runtime. Problems will have different numbers of materials, each of which contain different models and data – e.g., solids carry a stress tensor, gases do not.

`FleCSI` provides for some of this flexibility. Sparse storage patterns enable efficient representations of multimaterial problems by not storing all data associated with each material in each cell. Ragged storage patterns allow for model datastructures of various sizes to coexist with minimal overhead.

The way that `FleCSI` 1.4 expresses this flexibility is at compile time. In order to reason about the fields registered with the data client, their dependencies between tasks, and the tasks themselves, the 1.4 version relies on macro expansion into template specializations. We note that `FleCSI` 2.0 aims to move much of the compile-time requirements to the runtime.

Under this constraint, developers (acting as the earliest of users of their own code) wrote new source code to apply the overarching application to a new problem. When physics models had different enough methodologies, developers wrote entirely new applications. This resulted in an explosion in both the number of applications and the number of ways in which new code was added to the codebase.

To reign in these issues, we developed a structure for Ristra physics applications to adopt:

- a central **driver** to steer the application,
- a collection of physics **interfaces** that interact with the `FleCSI` mesh data structures, and
- a collection of **physics** modules that are effectively "zero-d" in that they don't interact with the mesh.

The structure clarifies where a developer should insert new code based on its function. Imposing the structure further fleshed-out patterns we had seen in datastructures developed to handle **M**ultiple **M**aterials with **M**ultiple **P**hysics models; we standardized these abstractions into what we called **M3P** datastructures and methods.

This added discipline made it easier to reason about the minimal set of items a user (read: developer at this time) needed to change for a new problem. We still had to write C++ code, but now it was delegated to a single header file (*material_types.h*) with an arguably simpler setup than when we started. Nevertheless, for new developers or those not familiar with C++ template metaprogramming – and especially for actual users that are not developers – writing this header file correctly (or debugging it when it was incorrect) was more difficult than we wanted. To that end, we developed a tool called `tide` that provides a connection between a user's Lua input file and the header file by generating said header file at "runtime."

## *Driver*

To the extent possible, our high-level multiphysics applications like `flecsale-mm` and `symphony` have a single driver. There are a few exceptions as we continue to bring more of the fragmented flock mentioned above into the fold.

In a traditional code, the driver could be thought of as the *main* function. `FleCSI`-based applications have a runtime based on the backend of choice (e.g., MPI, HPX, or Legion), so `FleCSI` itself owns *main* and calls into our driver. The driver's responsibilities include:

- parsing application-specific command line arguments,
- parsing the user's Lua input file,
- instantiating a mesh object and populating it with the contents of a mesh file on disk,
- instantiating any interface objects, and configuring them and their physics with user-set values from the input file,
- configuring material properties based on the input file and the *material_types.h* header,
- running the time loop of physics evolution,
- and calling out to restart or data dumps.

The driver makes no `FleCSI`-based calls. It knows nothing about the details of the mesh, the fields registered on the mesh, nor the tasks registered with the runtime. These details are handled by the interfaces, which expose methods to be called from the driver.

## *Interfaces*

Our usage of the term interface differs from the norm in C++ where one defines an abstract base class defining the API that children must implement. We avoid this form of dynamic polymorphism to avoid the potential costs associated with virtual table lookup, and any potential issues of registered field data/models being pointed to in a different memory space. Instead, we use interfaces as the abstraction layer between the high-level concepts needed to advance aspects of a simulation and the associated fields and tasks.

These concepts are illustrated through an example of one of our ALE hydro method interfaces – `maire_hydro_interface_t`. Figure 29 shows pseudocode for the constructor; it knows the fields it needs to grab handles to (e.g., scalar cell density, tensor velocity gradient, or M3P datastructures for materials) and default-initializes handles. The fields are known because they are registered elsewhere as shown in Figure 30. `flecsi_register_field` is a macro that expands to generate an instance of a complicated templated object that ensures hashes are created to uniquely identify the field.

Figure 29: Pseudocode of interface constructor.

```
maire_hydro_interface_t( mesh_handle_t * mesh_handle )
  : mesh_handle_ptr_( mesh_handle ),
...
    uc_grad_(
      flecsi_get_handle(*mesh_handle_ptr_, hydro,
                        cell_material_velocity_gradient,
                        tensor_t, sparse, 0)),
...
      all_states_(
        m3p::get_m3p_all_stateso(*mesh_handle_ptr_))
... {}
```

Figure 30: Pseudocode showing field registration.

```
namespace flecsale {
  namespace hydro {
    ...
    flecsi_register_field(
      mesh_t,                              // mesh type
      hydro,                               // namespace
      cell_material_velocity_gradient,     // field name
      tensor_t,                            // datatype
      sparse,                              // storage
      1,                                   // versions
      mesh_t::index_spaces_t::cells        // location
      );
    ...
  }
}
```

The `FleCSI` tasks called to advance the state use these private data handles of `maire_hydro_interface_t` to – in the case of the Legion backend – reason about data dependencies between tasks. Hiding these details behind the interface makes the driver cleaner and allows for easy reuse of parts of the interface without the need to obtain and pass in the complicted handles.

Continuing with the Maire hydro example, the timestepping loop within the driver looks as shown in Figure 31. Here, `hydro` is the instance of the interface class. `hydro.step` is a convenience method provided by the interface to be used in a driver focused solely on hydrodynamics, such as in the `flecsale-mm` application. `hydro.step` calls other public methods on the interface like `update_conserved_state(delta_t, future)` and `update_reaction(soln_time, delta_t)`. These methods are exposed so that other applications can pick and choose which components are needed at any given point in the operator split of its driver; `symphony`, which couples radiation from the `puno` code and hydro from the `flecsale-mm` code, does not use the `step` method, but rather picks things like `update_conserved_state` so it can interweave radiation updates to state variables.

The complexity of calling `FleCSI` tasks with all the handles is hidden behind these public methods of the interface. For example, `update_conserved_state`'s definition calls the registered task named `apply_update` as shown in Figure 32. Most of the private variables suffixed with _ are `FleCSI` data handles that are obtained during instantiation of the class as shown in the constructor in Figure 29.

The `apply_update` task is what uses the handle to the mesh to loop over mesh entities and modify field data. This is shown in Figure 33.

## *Physics Models*

As shown, interfaces call `FleCSI` tasks responsible for mesh-like operations on field data, including looping over mesh entities and materials. We define physics packages as a stateful class with associated parameters that implements a capability on a local level. Typically, "local level" means at the cell level, but in reality it means at a level that does not need to interact with the mesh. If things like gradients are needed, those should be constructed from the interface level and passed to the local physics package.

In the apply_update task in Figure 33, the h object is an instance of a hydro physics class; the multistate object from which h is obtained will be explained below in the M3P section. This example only showcases how h (local to the cell; note the cl index) is used to store data, e.g., the vector velocity field. However, physics package classes also

```
for (size_t num_steps = 0;
     (num_steps < max_steps && soln_time < final_time);
     ++num_steps) {
   // figure out the max possible time step
   ...

   // take a step
   hydro.step( soln_time, time_step, max_time_step );

   if (do_remap>0 && num_steps%do_remap==0) {
     if (remap_scheme != "euler")
       hydro.relax(remap_scheme, time_step, soln_time);
     hydro.remap(soln_time);
   }

   // update time
   soln_time += time_step;
   time_cnt++;

   // output the data if needed
   ...

   // now output the solution
   auto is_last_step = (num_steps==max_steps-1) ||
      (std::abs(soln_time-final_time) < epsilon);
   hydro.output( time_cnt, soln_time, is_last_step );
   hydro.conservation_sums(time_cnt);
}
```

Figure 31: Portions of the driver calling methods on the `maire_hydro_interface_t` interface instance `hydro`.

```
void update_conserved_state(real_t delta_t,
                            handle_t<int>& future) {
   flecsi_execute_task(
     apply_update,
     flecsale::hydro,
     index,
     *mesh_handle_ptr_,
     delta_t, dUdt_, all_stateso_, vfrac_,
     all_states_, uc_, pc_, dc_, ec_,
     alpha_, axis_, future
   );
 }
```

Figure 32: Example of an interface method that simply calls a `FleCSI` task. The use of the wrapping interface class and its private data handles simplifies the method call syntax.

```
void apply_update(
  flecsi_sp::utils::client_handle_r<mesh_t>        mesh,
  real_t                                           delta_t,
  flecsi_sp::utils::sparse_handle_r<flux_data_t> dUdt,
  m3p::all_material_sparse_handle_r_t              tstate0,
...
  handle_t<int> future
) {
...
  constexpr auto HYDRO = MP_KEY("hydro");
...
  // Using the cell residual, update the state
  for ( auto cl : mesh.cells( flecsi::owned ) ) {
    // get the cell volume
    const auto & cell_vol = cl->volume();
    // apply initial update
    for_each<m3p::num_materials>(tstate0, tstate,
        [&](auto && m, auto && state0, auto && state)
        {
            auto dUdt_cm = dUdt.at(cl,m);
               if ( !dUdt_cm.exists ) return;
            // restore old solution
            auto multistate0  = state0(cl,0);
            auto & multistate = state(cl,0);
            auto & h0 = multistate0.at(HYDRO);
            auto & h  = multistate[HYDRO];
...
            // get the cell state
            real_t et{h.internal_energy};
            for ( int d=0; d<num_dims; ++d )
              et += 0.5*h.velocity[d]*h.velocity[d];
...
        }); // for materials
...
  } // cells
} // apply_update
```

Figure 33: Example task from the interface for updating state.

have methods and through the M3P datastructures can query other physics packages. For example, the hydro physics classes store the local sound speed for calculations related to timestep size based on the Courant condition. The hydro physics class gets the soundspeed within its update_state_from_energy method by asking an – again local to the cell/material – EOS physics package to calculate it from the local density and specific internal energy.

## *M3P Datastructures and Methods*

Our datastructures and methods for handling multiple materials with multiple physics (M3P) underpin the capabilities of the above outlined structure. The goals of M3P are to enable communication between the physics packages without prior knowledge of specific instances existing and to simplify the workflow of adding a new physics capability. At a high level, the M3P capabilities use template metaprogramming techniques like SFINAE to construct a material model as a collection of physics models/packages (with associated class methods and parameters) known at compile time.

Each material exposes a key-item pair for the particular physics package, which registers itself with a common name. For example, all EOS physics packages (e.g., ideal_gas_t or eospac_t) inherit from a base EOS class that identifies its key as "eos". Using these keys (with a little extra M3P machinery) one can grab those models and use their data and methods as shown with the hydro model obtained from the multistate object with the HYDRO key in the apply_update task of Figure 33. These keys are also the means by which a physics package can ask for data/methods in other physics packages, as was described above with a hydro package calling an EOS package's methods to get the soundspeed. The hydro package doesn't need to know which type of EOS is available, just that one exists.

The collection of materials is placed into a container that reasons about its size at compile time. Similarly, the constexpr function template **for_each** shown in the apply_update task uses template metaprogramming to allow iteration over the different materials, which are different concrete types based on their physics models. **for_each** allows for application of a single callable (a lambda function in this case) on all materials to perform collective actions, such as updating their state or calculating total cell mass from the individual materials.

From the developer's perspective, the M3P abstraction unifies the way new physics capabilities are implemented. The developer need only implement methods and tasks at the interface level that iterate over cells and/or materials using the techniques mentioned, and the very localized physics package itself that can connect with other packages. If the capability is of a new type (i.e. not a "hydro", "eos", or other existing broad classes), the developer needs to create a new base physics class that defines the unique key used in the M3P datastructures.

The compile-time nature of these datastructures is required for the reasons outlined at the start of this section and also because `FleCSI` needs to reason about the size of the types to register with the runtime. This means a user who wants to run a problem still needs to write C++ code to declare these types. We have narrowed this down to a single file we call *material_types.h*. This file defines the available physics models (used to enable the compile-time key lookup), the specific material models, and the tuple collections for all the materials in the simulation. An example of a subset of the contents of *material_types.h* for a single material using a constant bulk modulus EOS, Lagrange hydro, a hypoelastic strength model coupled to an analytic power-law flow stress model is shown in Figure 34. Even with the convenience macros like MP_MATERIAL_TYPE that expand some of the template ugliness, this is overly cumbersome for developers to write just to run a different problem. We would get zero users if we asked them to write this as part of their input deck.

### *tide*

To hide the ugliness of *material_types.h* partial listing in Figure 34, we would like to have a better syntax for users to use in their problem set up. There are several patterns that appear when comparing different hand-written *material_types.h* files for various problems. This lends itself to code generation. Towards that end, we developed a tool called `tide`.

We already had Lua input files for setting physics properties such as user-written functions for initial conditions, or the ability to set gamma for an ideal gas EOS. These Lua input files needed to be used in conjunction with the *material_types.h* file to fully specify the problem. `tide` extends the capabilities of our Lua input files by allowing information in them to be used to generate the *material_types.h* file and compile it in at the last minute. This is not JIT compilation in the traditional sense – there is still an ahead-of-time compilation step that links the final executable. We simply try to hide this from the user so that for all purposes it appears that the Lua input file is what drives the simulation.

`tide` provides three general capabilities:

- a set of CMake utilities that allow for source-code parsing of available physics packages within the code, the appropriate build setup to compile as much code ahead of time as possible, and a gener-

Figure 34: Subset of example source code needed to write for a new problem.

```
// --- declare available physics --- //
using available_physics_t = MP_AVAILABLE_PHYSICS_TYPE (
    flecsale::basephysics::eos_t<real_t>,
    flecsale::basephysics::flow_stress_t<real_t>,
    flecsale::basephysics::hydro_t<real_t,
                              mesh_t::num_dimensions>,
    flecsale::basephysics::strength_t<real_t,
                                mesh_t::num_dimensions>,
    flecsale::basephysics::reaction_t<real_t> );
// --- begin materials definition --- //
using eos_constant_bulk_modulus_t =
  typename flecsale::physics::eos::constant_bulk_modulus_t<real_t>;
using hydro_lagrange_t =
  typename flecsale::physics::hydro::lagrange_t<real_t,
                                         mesh_t::num_dimensions>;
using hypoelastic_J2_t =
  typename flecsale::physics::strength::hypoelastic_J2_t<real_t,
                                               mesh_t::num_dimensions>;
using flow_stress_power_law_t =
  typename flecsale::physics::flow_stress::power_law_t<real_t>;
// - material 1 - //
using material1_t = MP_MATERIAL_TYPE (
       available_physics_t,
       hydro_lagrange_t,
       eos_constant_bulk_modulus_t,
       hypoelastic_J2_t,
       flow_stress_power_law_t);
using material1_state_t = MP_MULTISTATE_TYPE(material1_t);
...
using all_material_state_t = std::tuple<
      material1_state_t,
      material2_state_t,
      ...
      >;
```

ated shell script with appropriate settings to drive the final compilation and running,

- a user-facing input file validation of specified models against the database formed from parsing the source code with previous capability, and
- a user-facing approach to *material_types.h* source code generation.

The source code parsing uses libclang to look through the physics package source files for C++ user-defined attributes; we have been using [[ ristra :: physics ()]] , but the name is immaterial. When such an attribute is encountered on a physics package **class** or **struct**, information about that type is gathered and that type is added to a database of known physics models. The information gathered includes its base classes, template arguments, and the user-settable parameters and state of that physics package. This is represented internally as a Lua table, and is dumped to disk after all source files are parsed. These tables can be merged, as the structure of the table mimics that of the namespace structure used to define the classes in C++. Furthermore, documentation strings can be included in the attributes, which can be used to auto-generate the documentation from the C++ source code; this capability was strongly leveraged in the friendly-user class. Additional ideas for leveraging this attribute capability are to add constraints to certain parameters that can be checked when the input file is parsed; for example, specifying that the gamma in an ideal gas EOS must be greater than 1, and then this can be checked against the user's setting in a Lua input file.

The CMake utilities expose the source code parsing and generate the database of known physics models at build time. They also compile as much as possible ahead of time, generating and configuring installed files and a shell script. This shell script is what the user directly interacts with. It is set up to point to the installed files, one of which is a CMakeLists.txt file. The script takes as an argument the user's Lua input file and parses it for certain variables, such as those in Figure 35. In particular, the dim is used to set a #DEFINE macro, whereas the the models and materials tables are used to generate all the content in Figure 34 and more.

The use of strings will likely be removed; this is enabled via Lua's metatables functionality. At the input parsing stage, if there is a typo in a model name or the model name simply isn't in the database, the user is informed and suggested models are given. This input validation step is also where constraints placed on parameters (the gamma > 1 example) could be enforced.

Comparing the Lua code in Figure 35 with the *material_types.h* listing of Figure 34, the total set of models specified in Lua maps over

```
--single material problem
dim = 2
fp = "flecsale.physics."
models = {fp .. "hydro.lagrange_t",
          fp .. "eos.constant_bulk_modulus_t",
          fp .. "strength.hypoelastic_J2_t",
          fp .. "flow_stress.power_law_t"
}
materials = { [1] = models }
```

Figure 35: Example of additions to Lua input file needed to generate the listing of Figure 34 (and more).

to the available_physics_t, including the base classes of the concrete children specified in the input. Likewise, individual materials are constructed; here we are using "1" to identify the material, but this could easily be a string name. So, the simplified syntax of the Lua input file is used to generate the complicated C++ templated code. This is facilitated by using a text templating engine based on the mustache framework, called *lustache* whose source and license is included within `tide`.

The shell script mentioned above takes the generated *material_types.h* file plus the configured and installed CMake files and builds the final executable, which it then runs. The meat of the shell script is shown in Figure 36.

The end user would call the executable shell script (called `flecsale-mm`, here) with something like this

```
$ flecsale-mm -n 4 -i input.lua -m mesh_file
```

That single command validates the input file against the database, generates the *material_types.h* file if the input is valid, builds the final executable including the generated file (if needed), and runs the problem in parallel on four ranks. The compilation is hidden such that all the user sees is a short delay before the simulation runs (this is the purpose of the notify function in Figure 36). This much simplified problem setup allows for faster problem setup, a property important to both developers and users.

## 2 *The CS developer experience*

From a computer science perspective, two of FleCSI's overarching goals have been to reuse code where possible and appropriate and to create abstract interfaces to swappable dependencies. In practice, these principles have allowed for some interesting capabilities for users and developers alike, and have resulted in some tangible benefits for developing Ristra applications. Performance and portability have separately been notable concerns for the project. While per-

```
# setup the local build environment
mkdir -p ${local_build_dir}

# generate the source code file from the input
${tide} generate \
        -q \
        --db=${db_file} \
        -i ${input_file} \
        -t ${template_file} &
notify "Verifying your input file" $!
rebuild=true
# check to see if material_types.h has changed
# if not, then don't rebuild
if [ -f "${local_build_dir}/material_types.h" ]; then
  sha_old=$(sha1sum ${local_build_dir}/material_types.h
            | cut -f1 -d' ')
  sha_new=$(sha1sum material_types.h | cut -f1 -d' ')
  if [ "$sha_old" = "$sha_new" ]; then
    rebuild=false
    rm material_types.h dim
  fi
fi

if [ "$rebuild" = true ]; then
  mv material_types.h dim ${local_build_dir}

  # do the build
  cp ${cmake_lists} ${local_build_dir}
  pushd ${local_build_dir} > /dev/null
  cmake
    -DCMAKE_PREFIX_PATH="${cmake_prefix_path};${CMAKE_PREFIX_PATH}" \
    -DCMAKE_BUILD_TYPE=Release . > ${local_build_dir}/cmake.out 2>&1 &
  notify "Configuring your user experience" $!

  make -j VERBOSE=1 > ${local_build_dir}/make.out 2>&1 &
  notify "Customizing your user experience" $!

  popd > /dev/null
fi
```

Figure 36: The heavy-lifting part of the user-facing entry to a code built with `tide`.

formance and portability are often at odds with each other, we've striven to provide both. With respect to portability, FleCSI-based applications can be compiled and run on a variety of architectures. Our portability goals included running applications at scale on LANL's Trinity, a heterogenous system with Intel Haswell and Knights Landing processors, LLNL's Sierra, a GPU accelerated system with IBM Power9 CPUs and NVidia V100 GPUs, and Sandia's Astra system based on ARM CPUs. We've been successful in building and running various FleCSI applications on all three architectures, and can do so with zero architecture specific code changes.

## *Architectural Diversity*

Notable architectures:

- LANL Trinity, Trinitite – Haswell and KNL
  - MPI at ¼ Trinity (½ KNL partition), Legion multinode
- LLNL Sierra, RZAnsel – IBM Power9 + Nvidia V100 GPU
  - MPI at ¼ Sierra, Legion multinode
- SNL Astra, LANL Capulin – Thunder X2 ARM
  - MPI, full machine
- LANL Darwin – AMD Rome + MI60 GPU (WIP)

## *On-Node Code Portability*

One of FleCSI's primary goals has been to support code portability to a variety of computational architectures. Code portability to GPUs has been a focal point for FleCSI and the Ristra project, due to current and future GPU accelerated supercomputer acquistions within the Department of Energy. Targeting CUDA has been of particular interest in order that we may support execution on the Sierra supercomputer at LLNL.

### KOKKOS

Kokkos has provided FleCSI with the underlying structure on which we've been able to build a physics informed interface to a broad spectrum of computing architectures. With Kokkos, we've been able to rapidly prototype multiphysics applications capable of running on architectures supported by Kokkos. This not only gets us support on the set of hardware architectures that we're currently interested in targeting, but also the promise of support and utilization of future architectures through the efforts of the Kokkos development team. Kokkos provides an abstraction layer for writing performance portable kernels. FleCSI builds on top of Kokkos a physics-informed interface for application developers to target. Through Kokkos, the FleCSI parallel interface allows users to target an an abstract parallel device instead of a specific device or framework like CUDA, HIP, SYCL, or OpenMP. In turn, FleCSI is able to leverage all of the work that the Kokkos team has done to define a parallel interface and target specific frameworks or devices. Kokkos provides several key characteristics that allow us to provide consistent on-node parallelism in FleCSI

- C++ Compliant - One of the overarching goals of the Kokkos project is to minimize the barrier to entry for adoption. In practice, this is accomplished by defining Kokkos as a compile-time abstraction and ensuring that Kokkos can be compiled by a broad set of compilers. As a result, Kokkos does not impose any additional restrictions on FleCSI. FleCSI is able to leverage parallel APIs where they exist.

- Consistent in both data and execution model - Kokkos leverages APIs for a number of parallel interfaces, most of which provide different means of use. without Kokkos, FleCSI users would have to write disparate implementations of parallel code, likely with little opportunity for reuse. With a consistent data and execution model leveraged in FleCSI, developers can write code for data allocation and kernel execution once, and reuse simply by recompiling with a different Kokkos headers and library.

- Awareness and abstraction of API and Hardware details - Particularly when accelerators are involved, efficient use of hardware involves understanding details of hardware or APIs. For example, accessing a multi-dimensional array in memory may be most efficient in row-major order on a CPU when parallelized with OpenMP, but may be most efficiently accessed in column major oder when parallelized with CUDA on a GPU. Kokkos can hide some of these details behind its API, allowing the end user to focus on one API and its details instead of many.

Our use of Kokkos has allowed us to target and achieve performant on-node parallelism on disparate hardware. As a result, an application developer using FleCSI can write a task following FleCSI guidelines and reasonably expect that that task to run on CPUs and accelerators and to take advantage of the hardware parallelism they provide. FleCSI does not expose Kokkos interfaces directly, opting instead for a wrapper interface around Kokkos. The primary motivation for the creation of a wrapper has been to limit the scope of the dependency on Kokkos and to allow for the future possibility that wrapper layer could be reworked to wrap other on-node parallelism mechanisms (such as LLNL's RAJA).

#### KITSUNE

The Kitsune project at LANL is an effort to introduce parallel constructs in LLVM IR and use them to generate parallel code for a variety of parallel compute targets. Kitsune forwards language level loops to LLVM where they are translated into parallel intermediate representations. This allows the compiler to maintain information about the parallel constructs throughout the compiling process, potentially enabling additional optimizations while also reducing the complexity of the compilation process. With minimal changes to the above code example, Kitsune can compile the above example for a variety of devices, and has been used to parallelize Flecsale-MM, a multi-material hydrodynamics code based on FleCSI.

### *Distributed Computing*

As FleCSI has incorporated capabilities from other projects, a common theme of abstraction and generalization has emerged. A fundamental goal of minimizing reliance on any one specific technology has driven that theme, and this is best exemplified in our support for distributed computing backends. The FleCSI team works closely with and overlaps with the team behind the Legion programming model. In addition, we support MPI as a mechanism for distributed computing, and have ongoing efforts to evaluate HPX and Charm++ as additional backends.

#### LEGION

Legion is of particular interest to the FleCSI development team for several reasons. First, it provides fine grained task and data parallelism through a dynamic runtime that has the potential to be fundamentally superior to bulk synchronous methods used commonly in MPI. In addition, Legion provides intrinsic management of both data and execution across heterogenous architectures through a declarative interface. This allows a dynamic runtime to analyze requirements and optimally organize tasks given the resources available, both on-node and distributed. Legion also provides, in part through its Mapper interface, a mechanism for separating high level code from implementation details. In FleCSI, this means that application developers and even many FleCSI developers can be isolated from performance portability code, and that code for performance portability can be highly leveraged across many applications.

However, we don't want to tie FleCSI development solely to Legion. MPI is a much more mature, supported, and widely used technology. FleCSI has been developed to support MPI as both a first-class backend and as a supplemental backend to aid in migration. This backend abstraction provides for direct comparisons between backends in terms of features and performance.

In addition to a novel approach to distributed parallelism, Legion also provides mechanisms to ease performance portability.

- Kokkos Interoperability
- Device Memory Management - Legion handles data dependencies as part of its dynamic runtime capabilities. On supported devices with disparate memory spaces, this means that data are automatically migrated to the execution space where they are needed.
- Debugging - Legion supplies several tools for visualizing the structure and performance of a task-based application (Figures 37, 38, 39).

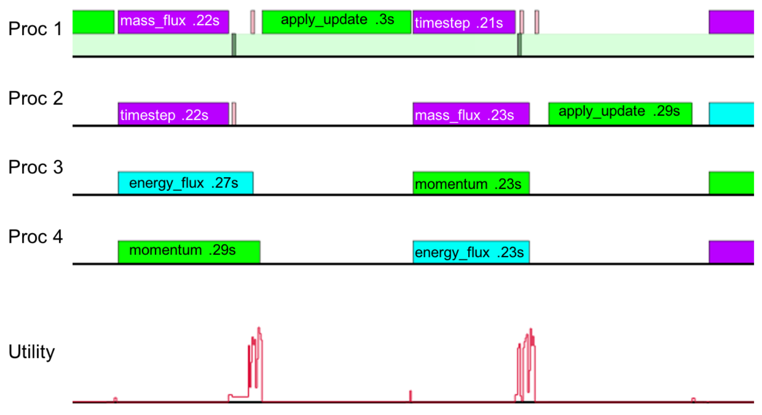


Figure 37: Profiling Legion Task Parallelism on 5 tasks

Legion has shown great promise for applications in the FleCSI Zoo. However, we have yet to fully realize the advantages that Legion provides. The constraining factors limiting our performance when using Legion fall into three categories.

The first category is limitations imposed by FleCSI. Legion's graph tracing capability provides a notable performance enhancement for applications with complex tasking and relatively small tasks. At a high level, Legion's graph tracing facility memoizes the task dependency structure in repeated multi-task workflows. This reduces runtime overhead by reducing the amount of task dependency analysis that the runtime has to perform on behalf of the application. This is particularly important for repeated task structures, which often occur in physics simulations. While future versions of FleCSI

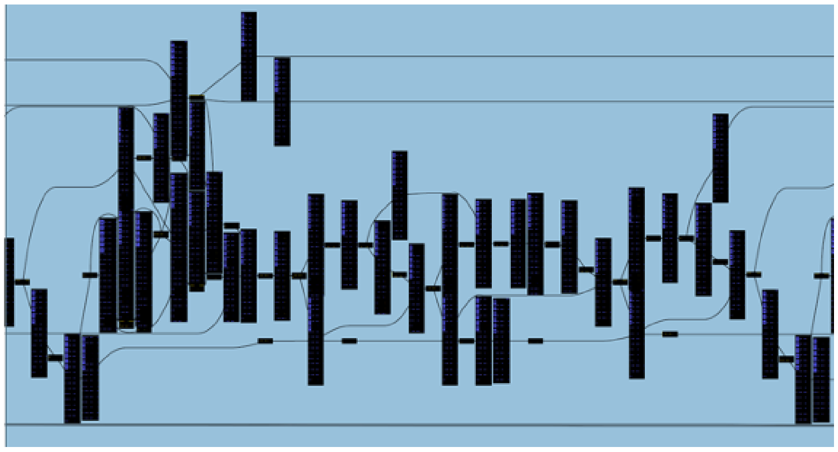


Figure 38: Profiling Legion Task Parallelism on 5 tasks

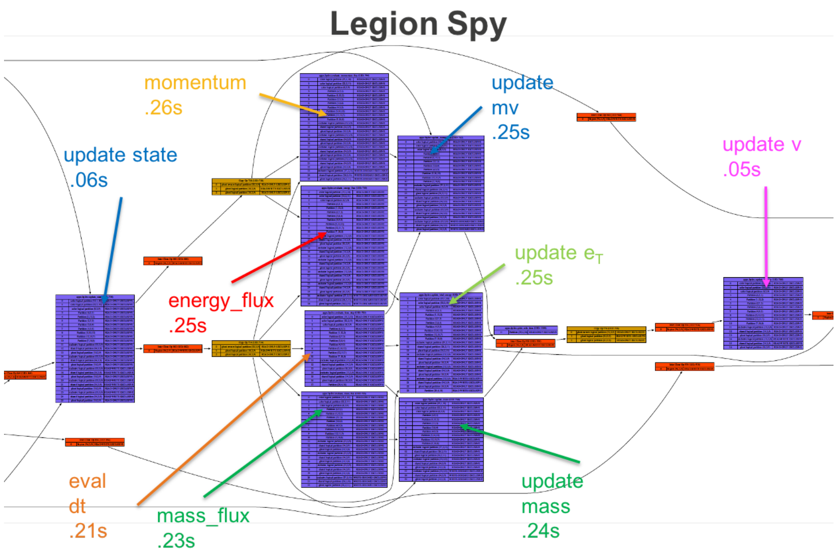


Figure 39: Using Legion Spy Data Dependency Analysis

will build upon this capability, the version used for applications in this milestone does not. As a result, in many cases our FleCSI-based applications are using Legion suboptimally and will likely improve once Legion graph tracing is utilized.

Legion also provides a concept called colors. Legion's coloring model allows for a mapping of domain partitions to processes other than 1:1. When a process manages multiple colors, effectively another level of parallelism through domain decomposition is exposed. This allows for more dynamic sharing of resources across partitions and less runtime overhead in both compute and memory footprint. As used in FleCSI 1.4, Legion is locked into a 1:1 mapping of colors to processes, so we are not able to fully utilize the dynamic resource sharing that Legion provides. We are currently paying an overhead penalty that will be reduced in future versions of FleCSI.

The second category is limitations that are imposed by the application in its use of Legion. Within applications built on top of FleCSI, we also currently exhibit some suboptimal behavior. Two common issues that commonly arise in FleCSI applications that hinder Legion performance are unnecessary data dependencies and synchronizing at unnecessary synchronization points. In the case of unnecessary data dependencies, an application developer may design a task with the intent to use a field and later end up not using it but still include the field as an unused dependency. In the case of the MPI backend, this has little impact to performance, as tasks are generally not asynchronously executed and fields are not copied or moved as a preamble to a task. For Legion however, an unused field can result in tasks that are functionally independent still being serialized relative to each other due to a constraint arising from the field. In addition, Legion may be required to copy the data represented by the field, even if it is not used, resulting in unnecessary runtime overhead.

Finally, the last category is limitations imposed by Legion itself. The most prominent example of Legion imposing a performance limiting constraint is the implementation for copy launchers. As part of a preamble or postamble of a task, Legion may have to copy data to or from data structures on which the task will operate or has operated. In some cases, the layout of the data to be copied may be complex, and Legion may do a suboptimal job at coalescing that data prior to transfer. In such cases, a copy launcher may move substantially more data than are necessary, resulting in reduced effective bandwidth. In pathological cases, this can result in a substantial performance impact. The Legion team is aware of this issue and has ongoing efforts to improve the efficiency of copy launchers in Legion.

### MPI

In order to mitigate potential risks with using Legion as a means for distributed parallelism and as a means of providing backwards portability while interfacing with or porting existing applications and libraries to the FleCSI framework, FleCSI can interface with MPI as a backend distributed parallelism framework. From a developer's standpoint, this offers several benefits. While developing new functionality in an application, the MPI backend provides a more familiar debugging experience, allowing for external tools like debuggers and profilers to interact with the application as the developer expects. For applications and libraries being ported to the FleCSI interface, it allows code to be adapted piecemeal, allowing the developer to ensure ported code works as expected at a finer granularity. From a testing standpoint, it gives developers a mechanism to assess how effectively they are using other backends through comparison.

### *Portability Across Layers of Parallelism*

From a developer's standpoint, taking advantage of multiple levels of parallelism across disparate architectures and programming paradigms is a highly attractive goal. Much of the development of FleCSI and its associated infrastructure has been in service of that goal. Figure 40 shows what code written to target FleCSI's portability looks like. Through Legion or MPI, FleCSI distributes tasks by partitioning a domain, in this case, a mesh. This figure shows two simple routines - the first computes a dot product of two fields across all cells through a reduction interface. The second computes a cell-wise axpy function for each cell in a mesh. While the calculations exhibited are simple, these are real code examples used in Puno as part of its conjugate gradient solver.

### *Visualization*

### *Difficulties*

C++17 and CUDA

Performance portability often imposes trade-offs, as constraints may differ between architectures. In particular, using C++17 with CUDA has imposed some difficult constraints. Given the long development timeframe for multiphysics codes, FleCSI needed to be as forward-looking as possible with respect to programming languages. Consequently, C++17 support was considered a requirement early on, with the presumption that compilers would catch up to our needs. Only recently has Nvidia provided even partial C++17 support for CUDA devices. As a temporary solution, we have used the Clang compiler to provide C++17 support for CUDA devices. Clang provides the ability to compile CUDA code, including C++17 features

```
double dot_product_cell(client_handle_r<mesh_t> mesh,
                               dense_handle_r<real_t> x,
                               dense_handle_r<real_t> y)
    {
      double sum(0);

      flecsi::reduceall(c, lsum, mesh.cells(flecsi::owned),
               flecsi::reducer::sum<real_t>(sum), "dot_product_cell") {
        lsum += x(c)*y(c);
      };

      return sum;
    }

    ////////////////////////////////////////////////////////////
    // z = alpha*x+y;
    void axpy_cell(client_handle_r<mesh_t> mesh,
                   real_t alpha,
                   dense_handle_r<real_t> x,
                   dense_handle_r<real_t> y,
                   dense_handle_rw<real_t> z)
    {
      flecsi::forall(c, mesh.cells(flecsi::owned), "axpy_cell") {
        z(c) = alpha*x(c) + y(c);
    };//c
```

Figure 40: Portable linear algebra functions in Puno.

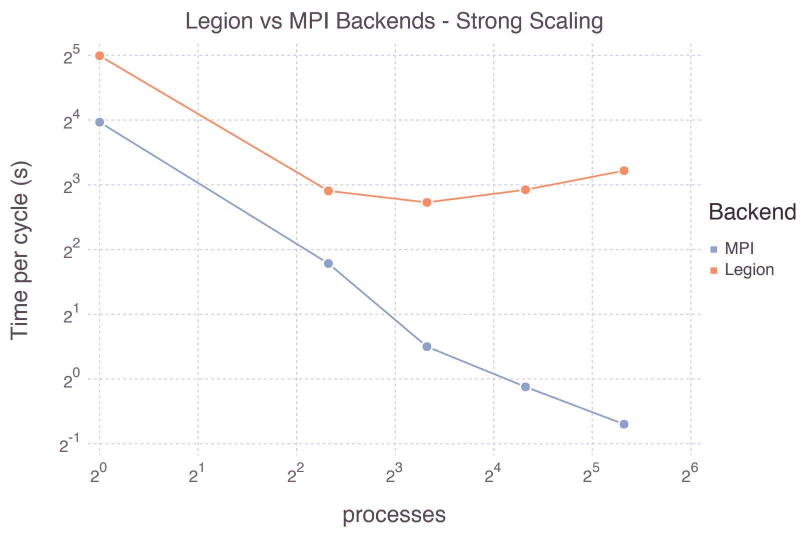


Figure 41: Strong Scaling Perforamance of Legion vs MPI backends on LLNL's Sierra

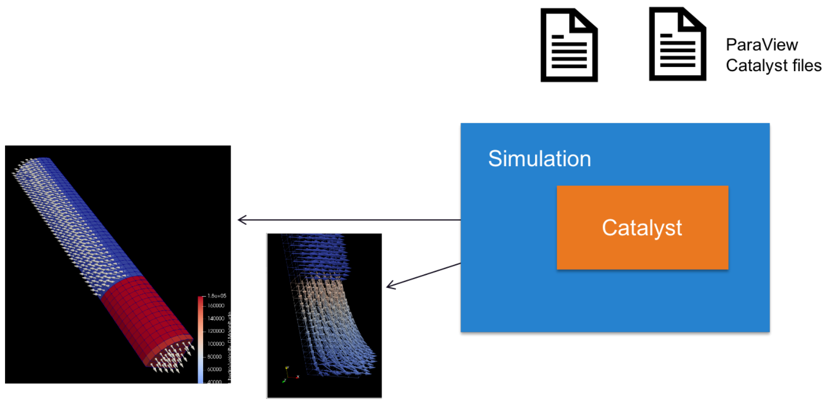


Figure 42: Paraview Catalyst Workflow

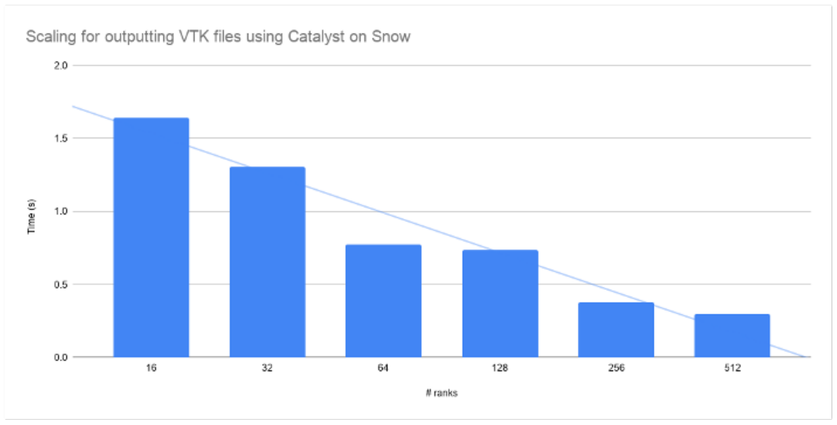
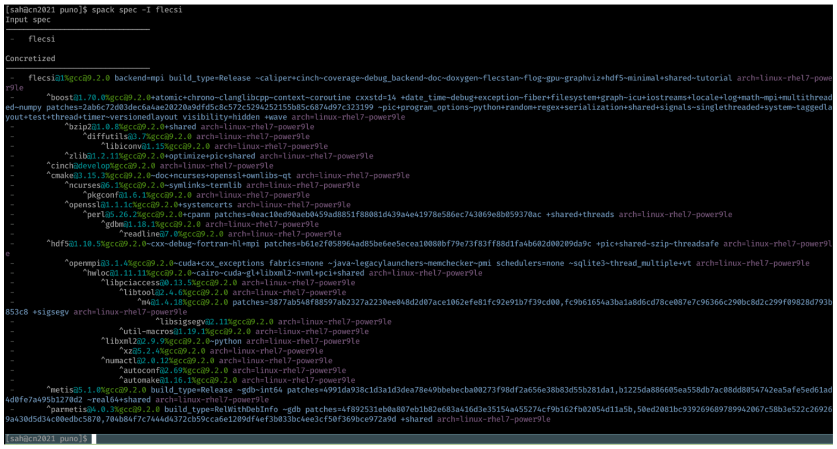


Figure 43: Paraview Catalyst Scaling

inside CUDA kernels. However, we have had to invest substantial effort building the infrastructure to compile FleCSI and its dependencies with Clang as both host and device compiler. This approach has pushed us closer to the standard workflow for AMD GPUs. As a result, we're well-poised to take advantage of AMD GPU Architectures with little change to our code infrastructure.

Due to the reliance on external dependencies for some of the underlying capabilities in FleCSI based applications, building up an environment for developing FleCSI based applications can be complex. This is particularly true on advanced architectures, which may require additional device libraries and compilers (CUDA for example). To mitigate this, the Ristra team has invested the time to build out a dependency management infrastructure through Spack (Figure 44). Spack provides developers a mechanism of programmatically defining a tree of dependencies and modifications to those dependencies to account for the special needs of FleCSI or derived applications. It also provides a mechanism for deploying those dependencies to users through the Environment Modules system found on most HPC platforms. As a result, user environments are more consistent and more easily maintained, while developers can more rapidly iterate on dependency additions and modifications.



```
[sah@cn2021 puno]$ spack spec -I flecsi
Input spec
--------------------------------
 -   flecsi

Concretized
--------------------------------
 -   flecsi@1%gcc@9.2.0 backend=mpi build_type=Release ~caliper+cinch~coverage~debug_backend~doc~doxygen~flecstan~flog~gpu~graphviz+hdf5~minimal+shared~tutorial arch=linux-rhel7-powe
r9le
 -       ^boost@1.70.0%gcc@9.2.0+atomic+chrono~clanglibcpp~context~coroutine cxxstd=14 +date_time~debug+exception~fiber+filesystem+graph~icu+iostreams+locale+log+math~mpi+multithread
ed~numpy patches=2ab6c72d03dec6a4ae20220a9dfd5c8c572c5294252155b85c6874d97c323199 ~pic+program_options~python+random+regex+serialization+shared+signals~singlethreaded+system~taggedla
yout+test+thread+timer~versionedlayout visibility=hidden +wave arch=linux-rhel7-power9le
 -           ^bzip2@1.0.8%gcc@9.2.0+shared arch=linux-rhel7-power9le
 -               ^diffutils@3.7%gcc@9.2.0 arch=linux-rhel7-power9le
 -                   ^libiconv@1.15%gcc@9.2.0 arch=linux-rhel7-power9le
 -           ^zlib@1.2.11%gcc@9.2.0+optimize+pic+shared arch=linux-rhel7-power9le
 -       ^cinch@develop%gcc@9.2.0 arch=linux-rhel7-power9le
 -       ^cmake@3.15.3%gcc@9.2.0~doc+ncurses+openssl+ownlibs~qt arch=linux-rhel7-power9le
 -           ^ncurses@6.1%gcc@9.2.0~symlinks~termlib arch=linux-rhel7-power9le
 -               ^pkgconf@1.6.1%gcc@9.2.0 arch=linux-rhel7-power9le
 -           ^openssl@1.1.1c%gcc@9.2.0+systemcerts arch=linux-rhel7-power9le
 -               ^perl@5.26.2%gcc@9.2.0+cpanm patches=0eac10ed90aeb0459ad8851f88081d439a4e41978e586ec743069e8b059370ac +shared+threads arch=linux-rhel7-power9le
 -                   ^gdbm@1.18.1%gcc@9.2.0 arch=linux-rhel7-power9le
 -                       ^readline@7.0%gcc@9.2.0 arch=linux-rhel7-power9le
 -       ^hdf5@1.10.5%gcc@9.2.0~cxx~debug~fortran~hl+mpi patches=b61e2f058964ad85be6ee5ecea10080bf79e73f83ff88d1fa4b602d00209da9c +pic+shared~szip~threadsafe arch=linux-rhel7-power9l
e
 -           ^openmpi@3.1.4%gcc@9.2.0~cuda+cxx_exceptions fabrics=none ~java~legacylaunchers~memchecker~pmi schedulers=none ~sqlite3~thread_multiple+vt arch=linux-rhel7-power9le
 -               ^hwloc@1.11.11%gcc@9.2.0~cairo~cuda~gl+libxml2~nvml+pci+shared arch=linux-rhel7-power9le
 -                   ^libpciaccess@0.13.5%gcc@9.2.0 arch=linux-rhel7-power9le
 -                       ^libtool@2.4.6%gcc@9.2.0 arch=linux-rhel7-power9le
 -                           ^m4@1.4.18%gcc@9.2.0 patches=3877ab548f88597ab2327a2230ee048d2d07ace1062efe81fc92e91b7f39cd00,fc9b61654a3ba1a8d6cd78ce087e7c96366c290bc8d2c299f09828d793b
853c8 +sigsegv arch=linux-rhel7-power9le
 -                               ^libsigsegv@2.11%gcc@9.2.0 arch=linux-rhel7-power9le
 -                       ^util-macros@1.19.1%gcc@9.2.0 arch=linux-rhel7-power9le
 -                   ^libxml2@2.9.9%gcc@9.2.0~python arch=linux-rhel7-power9le
 -                       ^xz@5.2.4%gcc@9.2.0 arch=linux-rhel7-power9le
 -                   ^numactl@2.0.12%gcc@9.2.0 arch=linux-rhel7-power9le
 -                       ^autoconf@2.69%gcc@9.2.0 arch=linux-rhel7-power9le
 -                       ^automake@1.16.1%gcc@9.2.0 arch=linux-rhel7-power9le
 -       ^metis@5.1.0%gcc@9.2.0 build_type=Release ~gdb~int64 patches=4991da938c1d3a1d3dea78e49bbebecba00273f98df2a656e38b83d55b281da1,b1225da886605ea558db7ac08dd8054742ea5afe5ed61ad
4d0fe7a495b1270d2 ~real64+shared arch=linux-rhel7-power9le
 -       ^parmetis@4.0.3%gcc@9.2.0 build_type=RelWithDebInfo ~gdb patches=4f892531eb0a807eb1b82e683a416d3e35154a455274cf9b162fb02054d11a5b,50ed2081bc939269689789942067c58b3e522c26926
9a430d5d34c00edbc5870,704b84f7c7444d4372cb59cca6e1209df4ef3b033bc4ee3cf50f369bce972a9d +shared arch=linux-rhel7-power9le

[sah@cn2021 puno]$
```

Figure 44: Flecsi Dependency Tree from Spack

## 3 *The user experience*

To assess the progress of *Ristra* codes, a workshop was recently held with several potential end users. Results and feedback from that event are discussed in an accompanying set of slides.

## 4 *A zoo of* FleCSI-*based codes*

One of the *Ristra* project's goals was to lower the barrier to entry of developing a computational physics code in a modern task-parallel style by providing a high-level abstract execution model, together with a data model that can be specialized to the needs of specific physics domains. *FleCSI* also provides an element of risk mitigation relative to the direct adoption of still-maturing task parallel programming models such as *Legion* or *HPX*, for example, since it provides the reliable option of a traditional *MPI* backend. It is encouraging that a number of *FleCSI*-based codes have been initiated over the last five years.

### *FleCSPH*

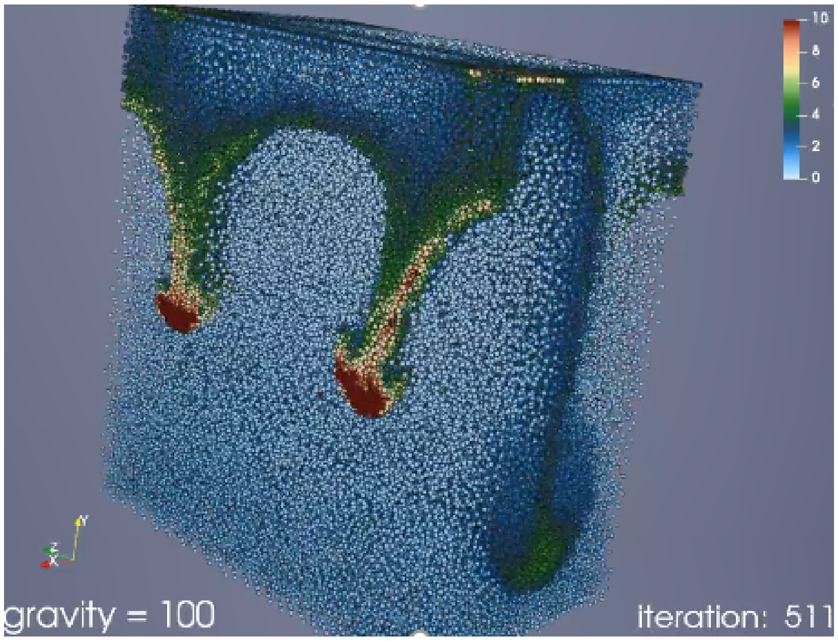


Figure 45: Gravity-driven Rayleigh-Taylor instability from a *FleCSPH* simulation

*FleCSPH* is a *FleCSI*-based smoothed particle hydrodynamics code that is being used for astrophysical research with LDRD funding (simulation pictured, Figure 45). The development team have worked closely with the *FleCSI* team to co-design new *FleCSI* topologies (compile-time specializations) appropriate to particle-based codes. In particular, the *tree* topology that is new in *FleCSI 2.0* grew directly from this collaboration. This topology is currently being used for early explorations of AMR in *FleCSI*-based codes.

## *MPAS-OCEAN*

The Model for Prediction Across Scales (MPAS) is a collaborative project for developing atmosphere, ocean and other earth-system simulation components for use in climate, regional climate and weather studies. *MPAS-Ocean* is a code designed for the simulation of the ocean system from time scales of months to millenia and spatial scales from sub 1 km to global circulations. With ECP support, a task-parallel version is being investigated. After initially implementing directly on *Legion*, the team decided on *FleCSI* as a more expedient path and started the *MPAS-O-FleCSI* code in collaboration with the *FleCSI* team. The coupling code is now working in prototype mode with interfaces for ocean, sea ice, atmosphere, and land in place together with time management and merge-average-remap capabilities. The next phase will be provided by *FleCSI 2.0* will enable multi-mesh capability and allow for investigation of increased task parallelism through breaking apart existing modules into smaller components (e.g. ocean column physics, ice pack physics).

## *CartaBlanca++*

*CartaBlanca++* is a *FleCSI*-based software environment for prototyping physical models and simulation of a wide variety of physical systems, using an ALE finite-volume method and the Material Point Method (MPM) and the Dual Domain Material Point (DDMP) method (simulation pictured, Figure 46).

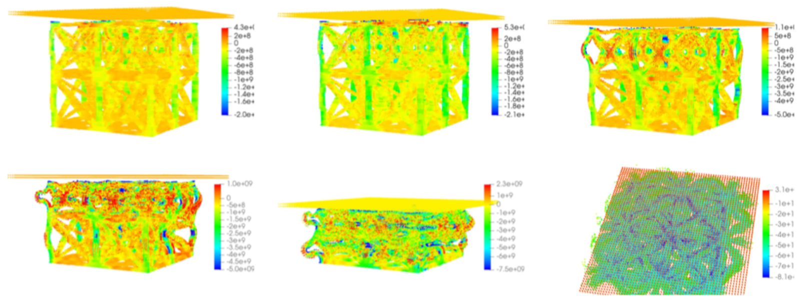


Figure 46: Deformation of a 3-D printed lattice calculated using *CartaBlanca++* with MPM. Particles are colored by the stress in the vertical direction.

It is being used in a number of application arrays including additive manufacturing (see figure) and in fracture and fragmentation field studies under the JMP program.

**Part V**

# Preparing for the future

## 1 Legion and task parallelism

### *Legion*

Scientific applications, and multi-physics applications in particular, will face significant challenges in realizing sustained performance on exascale systems. Increasing hardware specialization, power and cost constraints will result in exascale systems with order billion-way concurrency, a growing gap between memory and network latency and floating point performance, heterogeneity in both processing and memory capabilities and more dynamic performance characteristics due to power capping and highly tapered network topologies. Achieving sustained exaflop performance requires significant advances in latency hiding, minimizing data movement, and the ability to extract additional levels of parallelism from applications. An additional challenge is that these applications will need to achieve this level of performance on multiple exascale architectures. The Legion programming system [2] is well positioned to address these challenges. Legion provides high-level abstractions for application and middleware developers to describe properties of the data on which they compute, enabling the underlying runtime to automate many aspects of achieving high performance, such as extracting task- and data-level parallelism. Legion is designed to automate details of scheduling tasks and data movement (performance optimization) and separates the specification of tasks and data from the mapping onto a machine (performance portability).

How efficiently a supercomputer executes a simulation is directly related to its programming system—the system that tells the supercomputer which tasks to complete when. All supercomputers rely on parallelism, which allows the machine to perform multiple tasks at the same time, but the process of determining which tasks should run at which time is overwhelmingly complicated. Currently, that burden is placed on the those developing applications.

Figure 47 gives a graphical picture of a small supercomputing application that illustrates the complexity of the task scheduling problem. Each box is a task that must be scheduled at a time different from all of the boxes it is connected to. Satisfying the depicted constraints by hand, plus other constraints not illustrated (such as ensuring correct data movement and that tasks scheduled together do not require too many total resources) generally means adopting simple strategies that leave a great deal of potential performance untapped.

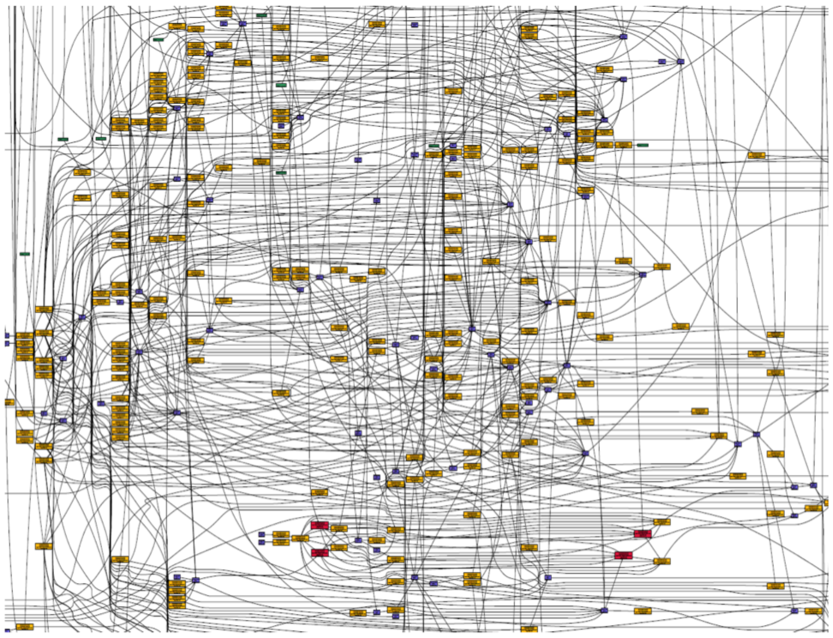


Figure 47: A task dependency diagram of a very small high-performance computing application that requires the application developer to manage and optimally schedule these tasks. Task scheduling can quickly become overwhelming for an individual, especially as the size of the application increases. Each box represents a piece of independent work (a task), with data flowing from left to right, and connecting lines represent data flowing from one task to another.

Adding to the application developer's burden is the fact that they are scheduling these tasks across a variety of supercomputing architectures, including many-core processors, graphics processing units (GPU), and central processing units (CPU), which are distributed across thousands to ten thousands of compute nodes. It's a massive amount of parallelism to track, let alone optimize for efficiency.

As the complexity of the application grows, the developer must spend more time recoding the application to perform well on each new computing system. This diagram illustrates a much larger number of independent tasks, represented by blocks in the same vertical column. These independent tasks must be scheduled to run at the same time to ensure optimal performance. Additionally, task placement must be optimized to minimize data movement (Figure 48).

The idea that a human (application developer) can optimize these tasks with high fidelity to fully realize the computing potential of supercomputers is clearly problematic. Application developers must put forth a heroic level of effort in writing application code, performance analysis, and performance tuning to realize the full capabilities of these systems. Furthermore, the cost of moving data within supercomputing architectures is starting to dominate the overall cost of computation, both in terms of power and performance. Managing

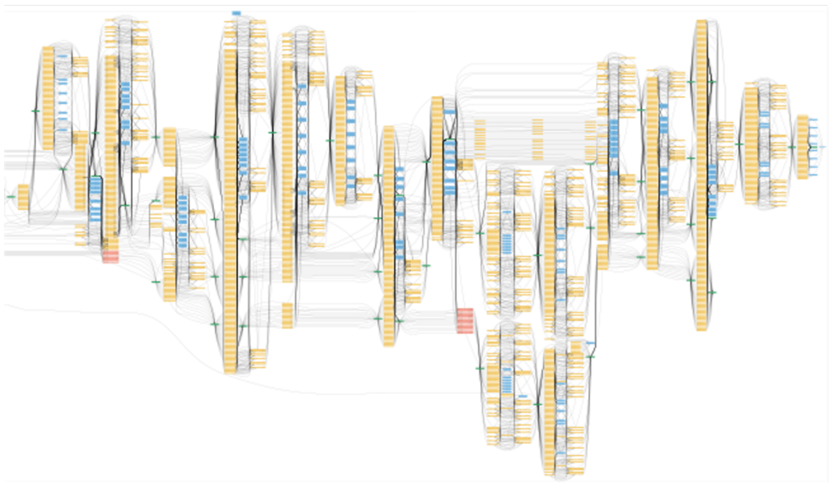


Figure 48: A task dependency diagram of a medium-sized high-performance computing application.

data movement within these supercomputers can be as challenging as managing tasks, and requiring a human to optimize data movement as well as tasks with high fidelity is simply not feasible. An automated solution for system programming is needed immediately.

Researchers at Los Alamos National Laboratory along with collaborators at Stanford University, NVIDIA, University of California-Davis, Sandia National Laboratories, and SLAC National Accelerator Laboratory have sought to address this problem. Legion automates task parallelism, data movement, and scheduling of execution, freeing up the application developer to focus on what is important for their domain of expertise and improving productivity of application development and the supercomputer. The capabilities of Legion and its impact across both scientific computing and machine learning was recently recognized with a 2020 R&D 100 award.

Legion solves three of the top programming system issues for next-generation exascale computing:

- High performance – Legion makes computing fast by automating task scheduling and data movement
- Performance portability – Legion automates scheduling across many kinds of machines (GPU, CPU, etc.) over many generations
- Programmability – Legion automates parallel execution of apparently sequential application code

In multi-physics applications the challenge of schedule task execution and data movement can be substantially more difficult than single physics. Recent work in the PSAAP-II Center at Stanford has demonstrated how Legion can manage this complexity in the Soleil-X multi-physics application which integrates radiation transport, fluid dynamics, and particle packages all written using the Legion programming system. Figure 49 illustrates this complexity and Legion's ability to see across multiple packages and manage data movement and task dependencies across these packages.

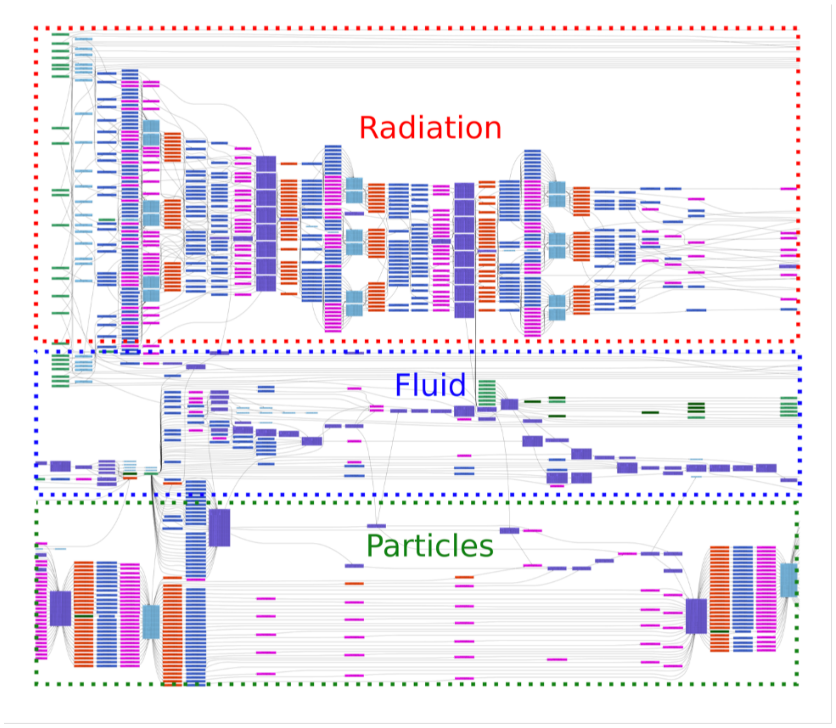


Figure 49: Data flow task graph produced by Legion for one time step in Soleil-X. The dashed boxes highlight the tasks associated with each physics solver.

The capability to manage data movement and task execution in complex multi-physics applications is the primary motivation for our use of Legion in the Ristra project. To this end, FleCSI was co-designed with the Legion programming system and adopted both its execution and data model. This has enabled FleCSI to establish a separation of concerns of application developers and computer scientists working within Ristra, freeing the application developer from the intricacies of data movement and task scheduling.

## *Experiences with task parallelism*

The *Ristra* team plan to produce a report on experiences and recommendations for task-parallel approaches for computational physics applications. Here we provide an example of the use of the software analysis tools available with *Legion* and the challenges associated with profiling in debugging in an asynchronous world.

USING LEGION TOOLS TO ANALYZE TASK PARALLELISM AND GENeral application performance

A Legion Prof analysis (Figure 50) of two iterations of the conjugate gradient solver in Puno reveals many things about its performance bottlenecks.

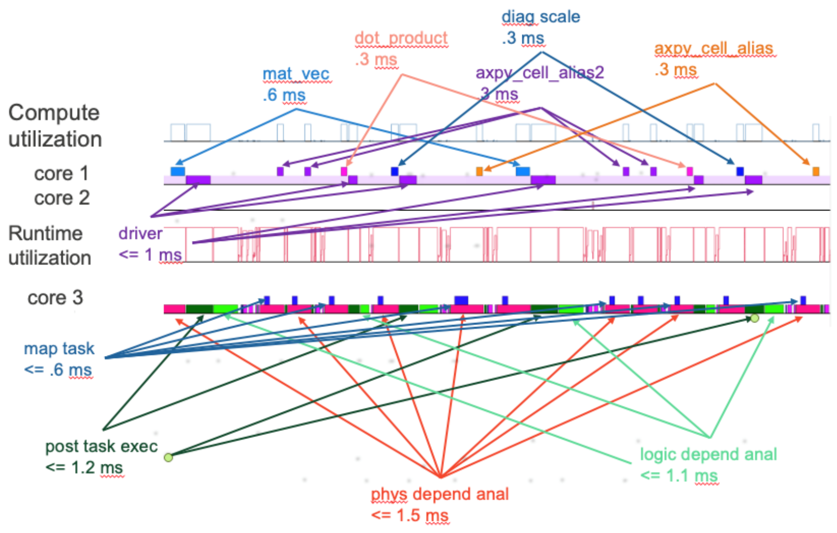


Figure 50: Legion Prof profiling analysis graph

To begin with, only one of the compute cores is being utilized. The application is not taking advantage of task parallelism. Runtime analysis tasks take 4X to 5X as long as computational tasks. Bigger tasks are needed to mask runtime overhead. Immediately after the mat_vec, dot_product, and diag_scale tasks, the driver (top-level task) switches on. This is because Puno is evaluating the reduction futures at the top level instead of passing them into the tasks that need them. This prevents the runtime from queuing up and completing analysis before the tasks themselves are run.

The reduction of dot_product is required to evaluate convergence, but the futures from mat_vect and diag_scale can trivially be passed into tasks instead of being evaluated at the top level. To determine if we can make bigger tasks by merging our small tasks and why we have no task parallelism will require a Legion Spy analysis.

The Legion Spy event graph analysis described in Figure 51 is shown at Adobe Acrobat's maximum zoom of 6400%. The event graph indicates all dependencies of events/tasks on each other (including futures). This is a small run for Puno, a few iterations and only two groups, but it is not possible to zoom in enough to read the output of this analysis. This should be improved.

The Legion Spy dataflow analysis (Figure 52) shows both application tasks and copy operations. The lines show data dependencies. For example, the bottom row shows a task that initializes data that is

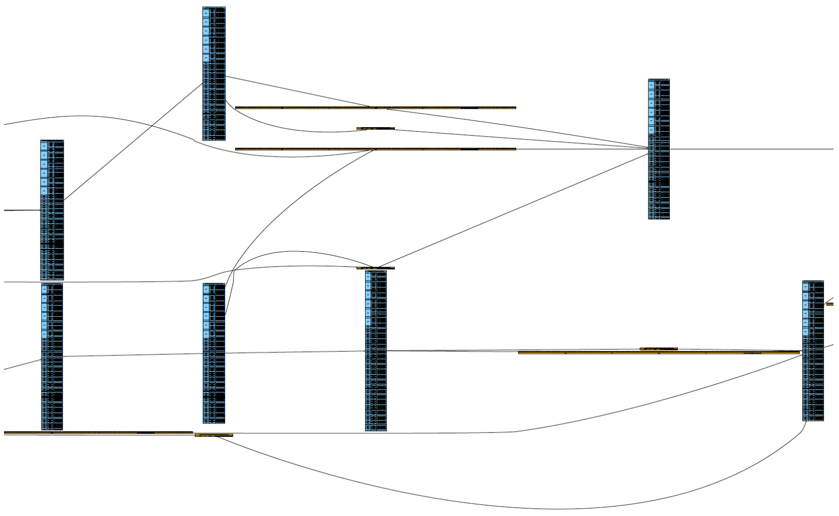


Figure 51: Legion Spy event graph

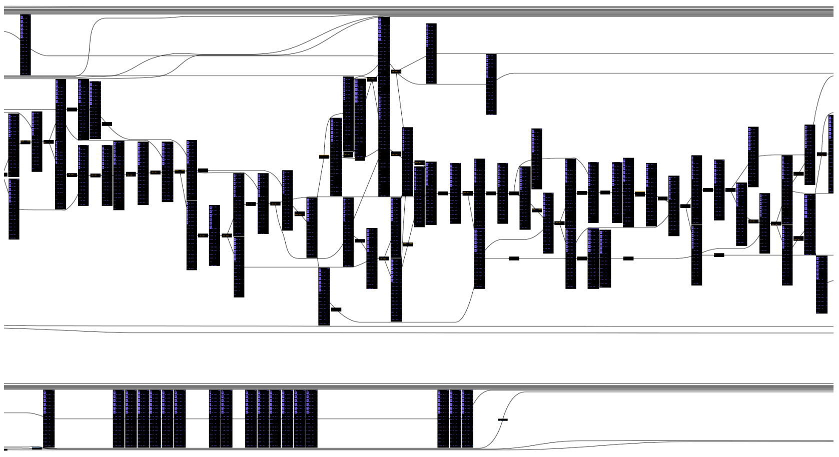


Figure 52: Legion Spy dataflow graph

never used. This task is an artifact that can be safely removed from the application and is an example of the ease of identifying orphaned code with Legion Spy.

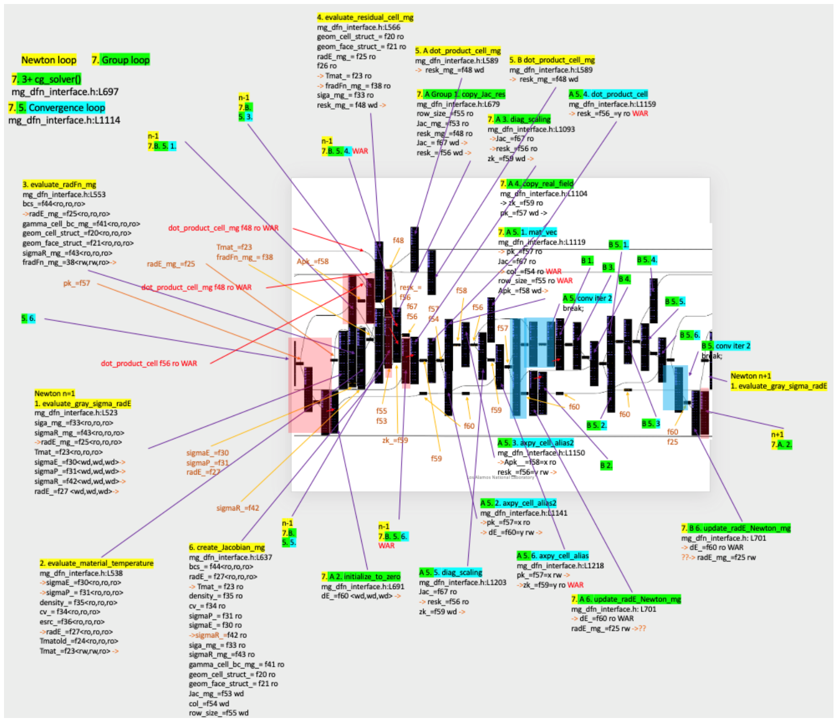

Newton loop
7. Group loop
7. 3+ cg_solver()
mg_dfn_interface.h:L697
7. 5. Convergence loop
mg_dfn_interface.h:L1114
4. evaluate_residual_cell_mg
3. evaluate_radFn_mg
dot_product_cell_mg f48 ro WAR
dot_product_cell f56 ro WAR
Newton n+1
1. evaluate_gray_sigma_radE
2. evaluate_material_temperature
6. create_Jacobian_mg
7. A 2. initialize_to_zero
7. B 6. update_radE_Newton_mg
7. A 6. update_radE_Newton_mg

Figure 53: Legion Spy dataflow graph before task merging

Figure 53 is a zoom-in of the dataflow showing task names, field numbers, and access permissions. This should be improved to show actual field names, but a careful comparison with the code can match numbers to names. From this analysis, we see that

- Each group has a write-after-read (WAR) dependency analysis that forces them to serialize and makes task parallelism impossible.
- Tasks with serialized dependencies (B must run after A; C must run after B; no other tasks use any of this data after A or before C) are candidates for task merging.

We have merged five tasks from the conjugate gradient solver into only two tasks as shown in the Figure 54 dataflow analysis. This reduces overhead and results in a 25% speedup overall for Puno (Figure 55).

It is not difficult to imagine the usefulness of these tools on a multi-physics code where no one developer is familiar with the entirety of the application. Indeed, these optimizations require knowledge of neither the physics nor the algorithms and could be carried out independent of the application development team.

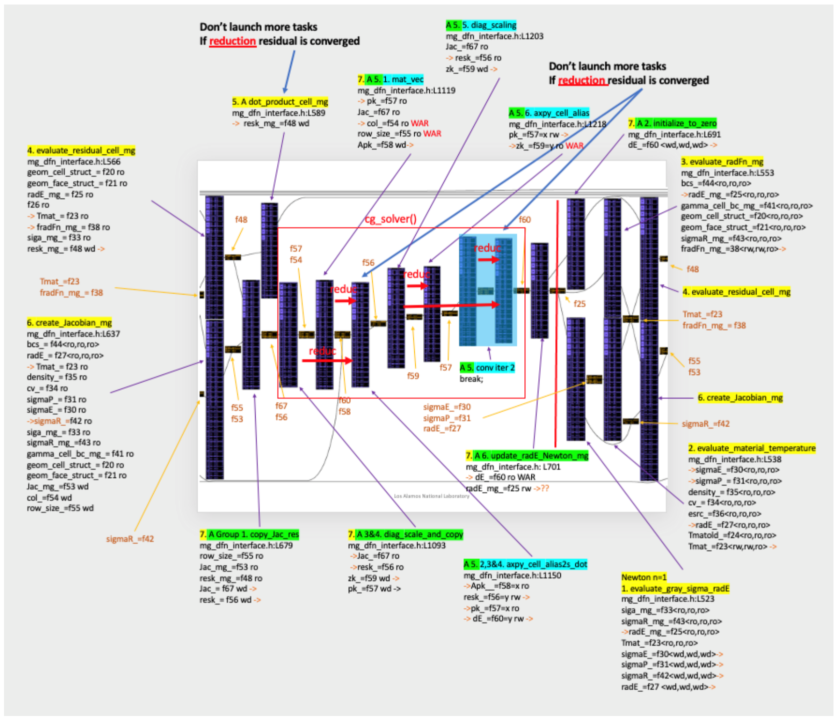


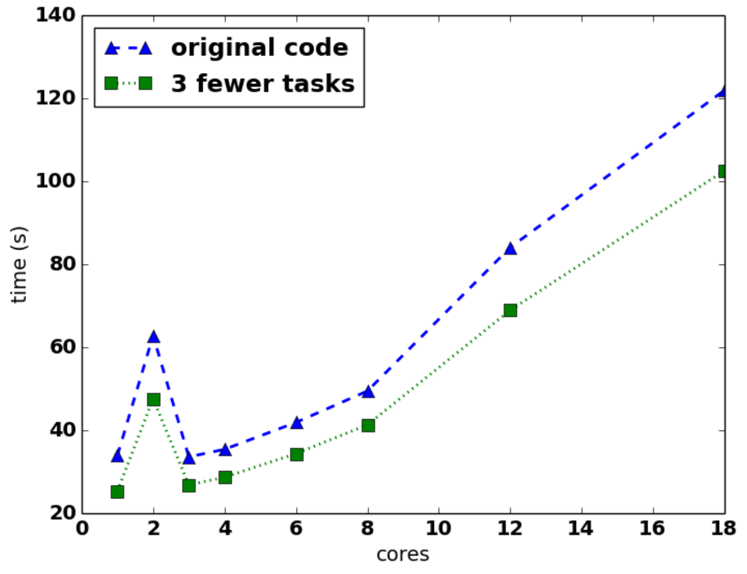
Figure 54: Legion Spy dataflow graph after task merging

**Other performance improvements for Puno**

- Only four of the tasks in the Newton loop shown above require the exchange of ghost data. Removing the unnecessary communication from the other tasks will speed up the code.
- Permissions are requested for data that is not actually used in a task. This prevents task parallelism from being discovered. These unused variables should be removed from the task signature.
- Enable task-parallelism between groups.

**Lessons learned and desired improvements**

- While FleCSI allows an application to run on either a bulk-synchronous or a task-based backend, it is unlikely for a single source to be performant on both.
  - Maximizing task parallelism can create more work and/or more cache misses for MPI.
  - Insufficient exposed task parallelism results in extra overhead for Legion with no benefits.
- A programmer thinking of MPI can break the FleCSI programming model in ways that only affect Legion.



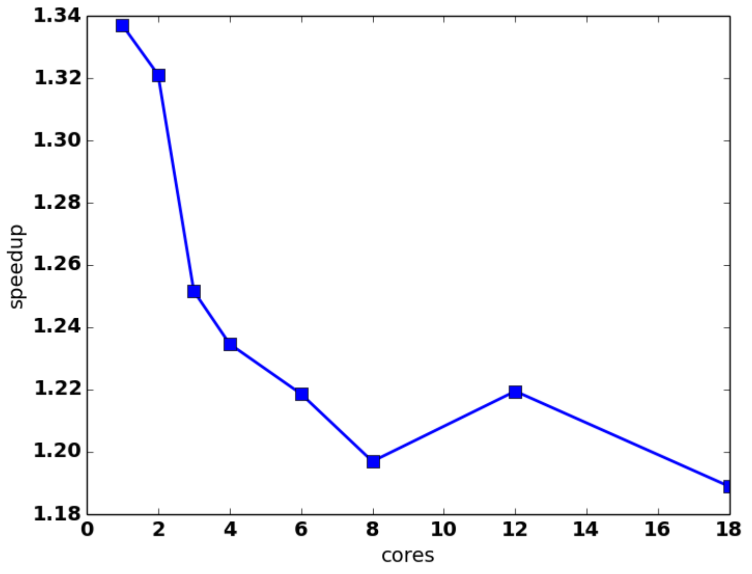


Figure 55: Performance graphs showing the impact of changing task granularity

  - Requesting permissions for variables that are not used but are already local can keep other tasks that actually use that data from running in parallel.
  - Modifying data without requesting write permission can yield incorrect results when task execution order is rescheduled.
  - Blocking on reduction results in the top-level task forces the application to be bulk-synchronous instead of task-parallel.

- As mentioned before, both of the Legion Spy analyses we used could use readability improvements.
- In creating scaling studies, we discovered bottlenecks in Legion itself (both are being addressed).
  - One degrades performance as we scale and prevents runs using hundreds of processes.
  - The other excessively penalizes performance for poorly chosen partitioning.
- Write-after-read dependence can prevent task parallelism from occurring. Storage is required to break this dependence, either in the application code or in the FleCSI mapper code.
- For some fields in some algorithms the ghost cell data can be calculated solely with local data. Unnecessary ghost copies introduced extra latency and slowed down the applications. FleCSI now has a way to opt out of ghost exchanges but an opt in strategy would encourage developers to pay attention to the costs of data exchange.
- Large tasks (>10 ms) in an application, even if there are seperable calculations, do not necessarily imply there will be an advantage to introducing task parallelism. For example, cache utilization could be affected or there could be a serial step in the overall calculation. In some cases, we found nearly equal performance on fewer ranks from a more task parallel implementation when both implementations were given an entire hardware node. When scaled out to hundreds of nodes, a factor of two reduction in the number of ranks employed, could still be a useful speedup.

## 2 *Kitsune & Tapir: Compiler Design, Parallelism, and Modern Architectures*

### *Kitsune & Tapir: Parallel-Aware Compilation*

With the slowing of Moore's Law and the end of Dennard scaling, systems have transitioned to using many-core and accelerator designs, and explicit parallel constructs have now become a given in what was once entirely sequential application code. However, the basic design of compilers has remained mostly unchanged throughout this transition. In particular, the optimization stages of compilation remain focused on improving sequential code performance with little to no awareness of the parallel semantics used today for programming node-level applications. As part of this milestone, we have explored the potential of expanding the compiler's awareness of parallelism throughout the full code generation pipeline. Before describing the specifics of our approach, we quickly provide some background on the fundamental design of today's compiler architectures and discuss where it falls short in capturing the nature of parallelism.

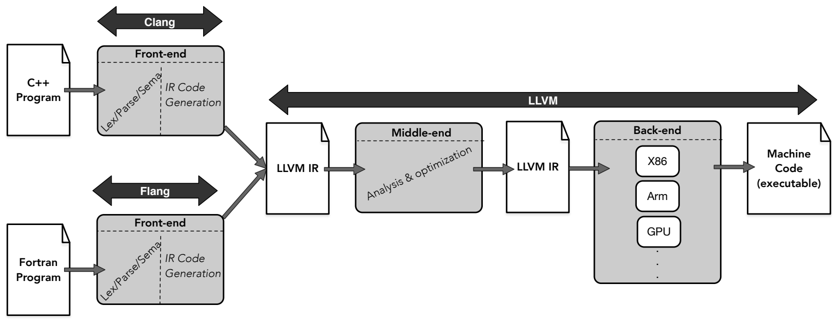


Figure 56: The LLVM Compiler Infrastructure [10] uses the classic pipeline design where the code passes through a series of transformations in the front-, middle-, and back-end stages. A language-centric front-end transforms the program into a common intermediate form that then proceeds into the middle stage. Within the middle stage, a series of architecture-independent passes analyze and optimize (transform) the IR given a general set of "best practices" and a given set of goals (e.g., performance, code size, etc.). After the optimization phase is complete, the IR continues into the back-end responsible for applying architecture-specific operations (e.g., register scheduling) and a very focused set of optimizations during the conversion of the IR to machine code.

A compiler for a sequential programming language, such as C or C++, can be viewed as three pipelined phases: a language-specific *front-end*, a language and architecture-independent "*middle-end*", and an architecture-specific *back-end*. The front-end parses the program syntax, performs semantic analysis checks, and translates the code into a language-independent form called the intermediate representation (IR). The IR enables the reuse of the remaining stages of the compiler across multiple programming languages. The middle-end consists of several stages that analyze and optimize the IR by transforming it into a more efficient form (in terms of performance, reducing code size, etc.). These passes are generally independent of the processor architecture and thus enable a layer of portability and generality. Finally, the back-end translates the optimized IR into machine code, and performs the necessary low-level target-dependent transformations and optimizations. For decades this design, as shown in Figure 56, worked in concert with Moore's Law to generate faster sequential code and hide the growing architectural complexity from application developers (e.g., vectorization and vector lengths, prefetching, etc.). While processors have become increasingly parallel, compilers have lagged behind, unable to analyze and optimize parallelism.

Instead of incorporating parallelism into the full compiler pipeline, today's designs have favored extending the front-end and leaving the majority of the middle and back-end stages unchanged. This design assumes that parallelism is a language-centric feature and assumes that sequential performance is still the principal goal at an architectural level — as shown in Figure 57, the design of Clang's [6] implementation of OpenMP follows this approach. Although addressing aspects of supporting parallelism, the implementation transforms the code into a sequential IR form with specific parallel runtime system function calls embedded within it (e.g., libomp). When the middle-end begins to process the IR, all parallel semantics are lost, and the OpenMP dependence has been "baked" into the code. In terms of parallelism, the IR code is no longer independent of the programming mechanisms. Adding any specific awareness of OpenMP into the middle- or back-end taints the generality of the intermediate form of the code.

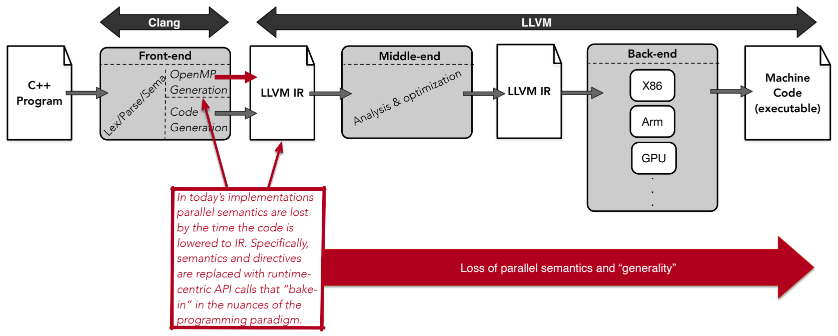

Clang
LLVM
C++ Program
Front-end
Lex/Parse/Sema
OpenMP Generation
Code Generation
LLVM IR
Middle-end
Analysis & optimization
LLVM IR
Back-end
X86
Arm
GPU
Machine Code (executable)
In today's implementations parallel semantics are lost by the time the code is lowered to IR. Specifically, semantics and directives are replaced with runtime-centric API calls that "bake-in" in the nuances of the programming paradigm.
Loss of parallel semantics and "generality"

Figure 57: Clang's current implementations of OpenMP "bakes" parallelism into the intermediate representation by inserting methodology and runtime-centric calls into the IR before the middle-end optimizations. The loss of parallel semantics impacts the generality of the design and limits flexibility and the potential of applying parallel-centric optimizations to the code.

To explore removing these limitations, we have modified both Clang and LLVM to make parallelism explicit within the implementation. Two components support this capability:

1. *Kitsune*:Provides extensions and modifications to Clang that address a portion of the shortcomings discussed above. We describe the functionality of Kitsune in the sections that follow.

2. *Tapir*: An extension to LLVM's intermediate representation (LLVM IR) to support the representation of parallel constructs in the form of fork-join operations. As we do not provide a detailed discussion of Tapir in this report, interested readers can refer to the following papers for more details [14, 15].

Both Kitsune and Tapir are packaged in an open-source fork of the full LLVM Project and both development code and releases are available via GitHub at `https://github.com/lanl/kitsune`. Our current implementation builds upon the version 10 release of LLVM/Clang.

This combination of Kitsune and Tapir maintains parallel semantics in the IR throughout the middle-end's passes. This design enables supported runtime targets to become a middle-end code generation target versus being inserted into the IR by the front-end. From our viewpoint, this maintains the design principle of separation of language and syntax choices from code analysis, optimization, and generation for the later stages of compilation. Figure 58 shows how this approach changes the design of the compiler pipeline. It allows for additional flexibility in code generation at both the software and hardware levels during the middle- and back-end stages. This can improve flexibility and portability and support specific enhancements via runtime libraries without requiring modifications to the front-end.

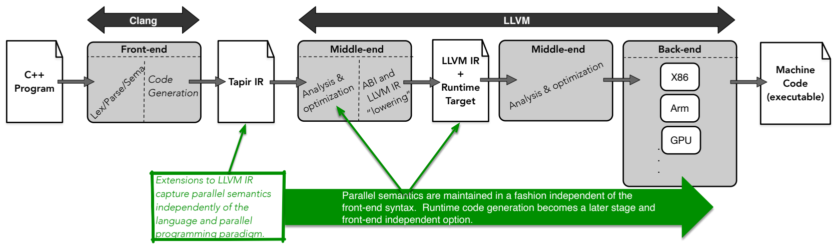

Clang
LLVM
C++ Program
Front-end
Lex/Parse/Sema
Code Generation
Tapir IR
Middle-end
Analysis & optimization
ABI and LLVM IR "lowering"
LLVM IR + Runtime Target
Middle-end
Analysis & optimization
Back-end
X86
Arm
GPU
Machine Code (executable)
Extensions to LLVM IR capture parallel semantics independently of the language and parallel programming paradigm.
Parallel semantics are maintained in a fashion independent of the front-end syntax. Runtime code generation becomes a later stage and front-end independent option.

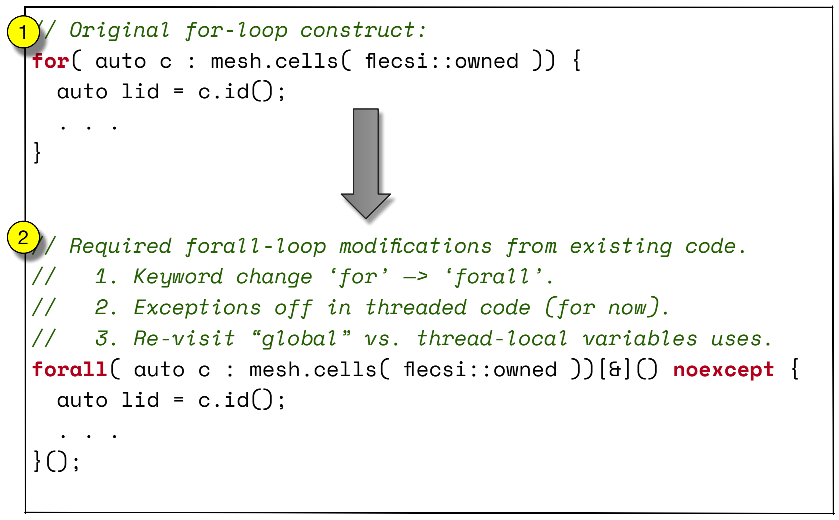
Figure 58: Kitsune and Tapir make changes to the front- and middle-ends of the LLVM compiler infrastructure to maintain an awareness of parallelism during compilation. This design allows the analysis and optimization stages to become aware of parallelism semantics and maintain a more general design versus a hard-coded set of mechanisms dependent upon a particular language or programming paradigm used in the front-end.

As part of our milestone efforts, we have explored the design, implementation, and use of the Kitsune+Tapir prototype compiler. The following sections present the specific features we explored, provide two example use cases, discuss the results and lessons learned, and discuss future directions. Our initial approach has focused on mechanisms that can have a minimal set of required changes to applications written in C++.

## THE `FORALL`-LOOP CONSTRUCT

During early design discussions with the FleCSI team, we determined that introducing a new keyword to flag the semantics of widely used data-parallel loops was likely to have a minimal impact on the codebase. As a result, Kitsune supports the `forall` keyword that follows C++'s existing `for`-loop statement. When not using Kitsune, this approach provided a straightforward path of using a simple macro to allow code to be compiled with traditional compilers. However, when compiling with Kitsune+Tapir the construct has an explicit set of semantics that the developer must be aware of:

- All loop iterations must be able to execute independently of all others.
- Developers should make no assumptions about the ordering of the iterations.



```
// Original for-loop construct:
for( auto c : mesh.cells( flecsi::owned )) {
  auto lid = c.id();
  . . .
}

// Required forall-loop modifications from existing code.
//   1. Keyword change 'for' -> 'forall'.
//   2. Exceptions off in threaded code (for now).
//   3. Re-visit "global" vs. thread-local variables uses.
forall( auto c : mesh.cells( flecsi::owned ))[&]() noexcept {
  auto lid = c.id();
  . . .
}();
```

Figure 59: The steps required to transform a C++ `for` loop into a Kitsune+Tapir supported `forall` loop. This example comes directly from the Flecsale application and is complicated by the main application code having exceptions enabled and therefore requires a more complex transformation to disable them within the loop.

- The developer is responsible for ensuring any (data) state outside of the loop body is read-only or safe.

- For best performance, C++ iterators used as loop induction variables should be random access.

- Currently, exceptions within the loop body (or code that surrounds the loop) must be disabled. This lack of support is primarily driven by exceptions in parallel code being poorly defined within the compilers implementation — a solution will require some careful implementation details and active efforts are underway to explore solutions.

Figure 59 show an example of the steps needed to transform a traditional C++ for-statement into a corresponding forall-statement. If the semantics are followed, the change can be as simple as replacing `for` with `forall` in either traditional C-style for loops or in C++ range-based loops. If exception handling is enabled, a more complex set of changes may be required. For example, the example in Figure 59 shows a transformation into a lambda expression to allow the use of the `noexcept` keyword.

### USING FORALL STATEMENTS

We have experimented with using forall-statements in the FleCSI-based *flecsale* and *flescale-mm* applications. In addition to the changes discussed above the standard Clang compile line requires the addition of a single argument, `-ftapir=<rt-target>`. This flag serves to enable the `forall` keyword and specifies which runtime will be targeted during compilation. At present we have support for, or implementing, the following runtime system targets:

- `cilk`: The Intel Cilk Plus runtime.
- `opencilk`: The new OpenCilk runtime.
- `qthreads`: The Qthreads runtime system.
- `cuda`: NVIDIA CUDA runtime and PTX code generation. (*in progress*)
- `kitcuda`: A wrapper around cuda meant to simplify code generation. (*in progress*)
- `openmp`: The OpenMP runtime library.
- `opencl`: The OpenCL runtime and SPRIV code generation. (*in progress*)
- `hip`: AMD's HIP layer and AMDGCN code generation. (*in progress*)
- `realm`: Legion's low-level runtime. (*in progress*)

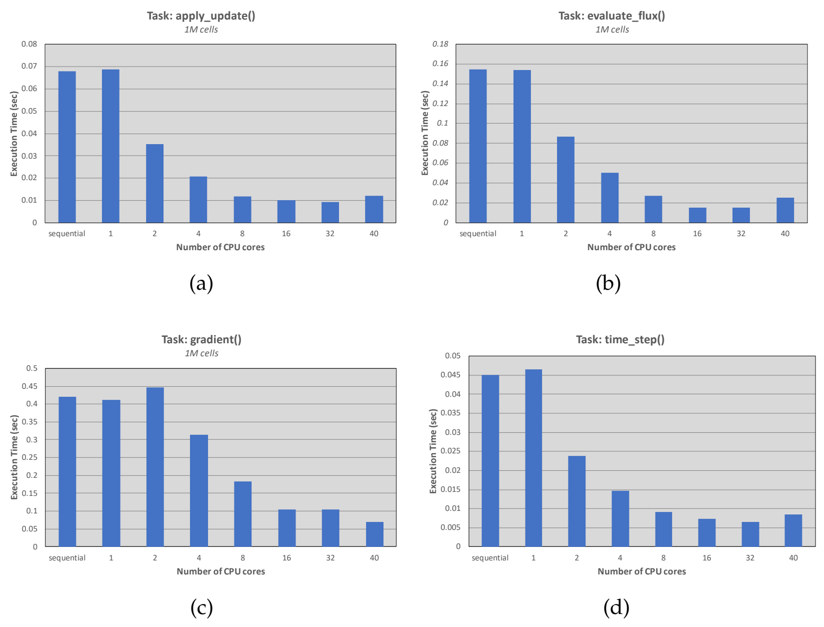


Figure 60: The performance of the `forall` loop constructs within the flecsale application on a many-core CPU system. The sequential time reported is the unthreaded application using Clang with -O3 optimizations enabled. The single threaded case is presented to reveal overheads from the runtime system and associated code generation mechanisms.

With these code changes and the new compiler flag, we have re-implemented four tasks within the flecsale applications to use the `forall` keyword. Figure 60 presents the results of running these applications using the `cilk` runtime target on 1, 2, ..., 40 cores. Importantly, this approach allows developers to pick a runtime system tailored to a target architecture or the specifics of an application's workload. Recall that the runtime target code is generated after the code has passed through the compiler's various optimization stages, in contrast to adding complexity to adding the code before these passes. Unfortunately, at the time of writing, we have not yet completed an analysis of the impact this has on the overall code generation characteristics of the flecsale and flecsale-mm applications. In the following section, we briefly discuss the changes to Clang that are required to support `forall` statements.

#### LOWERING `FORALL` TO TAPIR IR

Support `forall` requires a minimal introduction of new code into Clang. At a fundamental level, a new keyword and corresponding statement support can build directly on the same infrastructure used by `for` statements (including C++'s range-based loops). Although some of this code can be complex, it is reasonably straightforward to maintain and keep up to date with the main LLVM project. The one area where entirely new code is needed is the "*lowering*" of Clang's AST into the Tapir intermediate form. In this case, care must be taken to avoid the introduction of race conditions, support the appropriate handling of thread-local state, and maintain metadata for code analysis in later stages of compilation (e.g., type-based alias analysis). Overall, the required code to support `forall` is approximately 1,200 to 1,500 lines of code. Of that, roughly 600 lines are responsible for the IR generation.

The very generality of `forall` loop constructs is both their power and part of the complexity they introduce when used with advanced C++ code. This complexity complicates the compiler's analysis and optimization via overheads and potentially lost opportunities for better code generation and improved application performance. In the next section, we explore an alternative and more aggressive path for generating intermediate representations of parallel loops.

### *Kokkos-Centric Code Generation*

Kokkos, Raja, and Kitsune+Tapir share some similar goals: they all seek to capture parallel constructs via a combination of syntax and semantics. The fundamental difference is that Kokkos and Raja rely on C++ mechanisms to implement a code generation framework for an underlying target programming system (e.g., OpenMP, pthreads, CUDA, etc.). While this provides a path for portability, it potentially adds layers of complexity to the code in addition to the lost parallel semantics previously discussed. The C++ approach's advantage is its independence of any particular compiler, and one could argue, requires less effort to implement than a compiler. However, as discussed below, the compiler implementation path provides some distinct advantages. In addition, the wide adoption and design of LLVM certainly support production-level compiler efforts in ways that were not possible in the past. Following a similar set of motivations to the introduction of the `forall` keyword, we explored the potential of creating a *Kokkos-aware* version of Clang.

This approach allows Kokkos to be *explicitly* recognized by the compiler — skipping the need to introduce new keywords into C++. The implementation is achieved by recognizing Kokkos data types and then circumventing Clang's C++ code generation techniques by directly lowering the code to Tapir's parallel IR form. The fundamental structure of the IR is no different from the representation of `forall` loops. For code generation, Kitsune replaces a Kokkos `parallel_for` with a simple loop — skipping template expansion, the transformation of lambda expressions, and the generation of the targeted runtime or programming system. An example of a supported Kokkos `parallel_for` construct is presented in Figure 61.

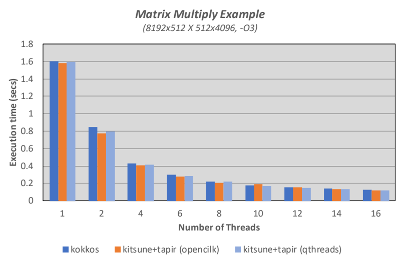
```
float *A = new float[VEC_SIZE];
float *B = new float[VEC_SIZE];
float *C = new float[VEC_SIZE];
. . .
// A simple paralell vector element sum.
Kokkos::parallel_for(VEC_SIZE, KOKKOS_LAMBDA(const int i) {
   C[i] = A[i] + B[i];
});
```

Figure 61: A example of the lambda form of the Kokkos `parallel_for` construct that is currently supported by Kitsune.

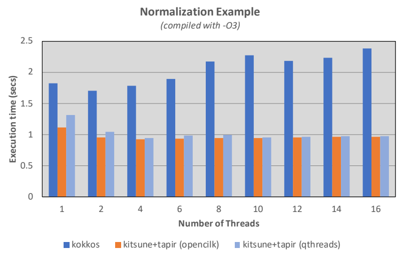
## USING KITSUNE'S KOKKOS MODE

Much like supporting `forall`, the Kokkos mode of compilation requires two flags. The first, `-fkokkos`, puts the front-end of the compiler into a mode of operation that understands the details of Kokkos constructs, data types, and semantics. The second flag enables Tapir support via the previously discussed `-ftapir` command-line option. At present, Kitsune only supports a limited subset of the possible forms that Kokkos can represent. However, this initial feature set has provided the opportunity to explore the impact on code and the internal processes inside the compiler.

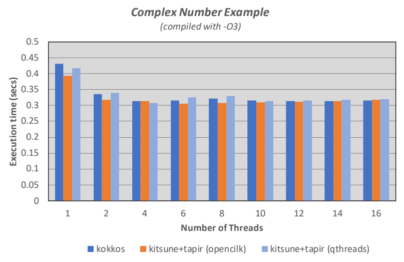
We used a small set of `parallel_for` examples to explore the performance of the resulting code. These examples do not capture the complexity of large scale applications but are suitable to explore different target architectures, runtime targets, optimizations, and additional features inside the compiler. The results of these benchmarks are shown in Figure 62. Fundamentally, these results highlight limitations of the underlying system architectures (e.g., memory bandwidth limits in the case of the vector addition example), some slight nuances in the behavior of the targeted runtime systems, and in many cases shows where the the compiler was able to apply the same optimizations to either code representation. However, there is a distinct advantage of the parallel form's explicit nature that did allow for better optimizations to be applied to the code. The clearest example of this occurs in the case of the "normalization" benchmark were the implementation of C++ lambdas, aggressive (early) inlining, early runtime transformation, and other details obscured the fact that a very expensive function call could be hoisted outside of the `parallel_for` body. Given that small (1-2%) improvements in the performance of a compiler's set of optimizations are viewed as a positive result, we believe there are benefits in further exploring analysis and optimization enhancements using LLVM and the Tapir IR.



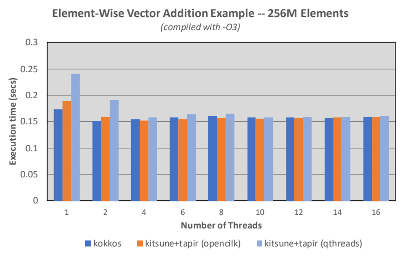
(a)



(b)



(c)



(d)

Figure 62: A set of simple Kokkos `parallel_for` benchmarks used to explore our initial steps of code optimization passes with the Tapir intermediate representaiton. Several of the benchmarks actually end up being constrained by the runtime system or charateristics of the underlying hardware (e.g., memory bandwidth). In some cases we see distinct advantages where traditional C++ inlining and complexity of the code structure generated by template metaprogramming end up at a disadvantage of the parallel representation. The "normalize" benchmark highlights one such case where inlining obscured the fact that an entire function call could be hoisted outside of the loop body.

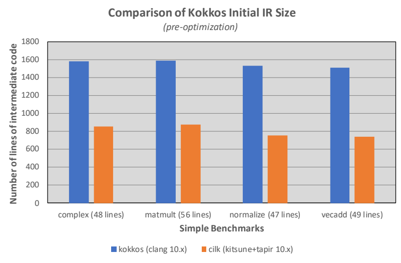
### LOWERING KOKKOS TO TAPIR IR

The changes to Clang to support Kokkos-centric compilation direct follow the motivations behind domain-specific approaches and techniques. Given that we maintain the original form of the C++ code, the changes require adding around 500 lines of code to Clang's IR generation library. The implementation also follows many similar steps as those used by the `forall` implementation. The overall process of replacing the lambda construct with an explicit loop, significantly reducing the complexity of the code sent to the compiler's middle stages.

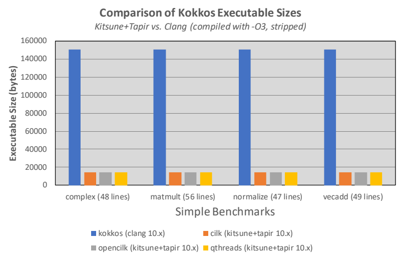
Many of the optimization passes in the compiler are complicated and expensive computations. This complexity directly impacts compile times, which reduces productivity in terms of developer time spent waiting for the compiler, the turn around time and resource utilization of continuous integration operations, and the amount of time the compiler has to search for potentially beneficial optimizations to apply.[1] Therefore, it is not surprising to find that the time required to compile a program is directly related to the number of lines of IR produced by the front-end. Figure 63 shows some comparisons between the full C++ path of compilation and the results when using the Tapir IR with Kitsune.

[1] Some compilers have the notion of an internal "*time quota*" that attempts to maintain productivity. After meeting the quota, the compiler stops optimizations and proceeds to code generation – often resulting in under-performing code.

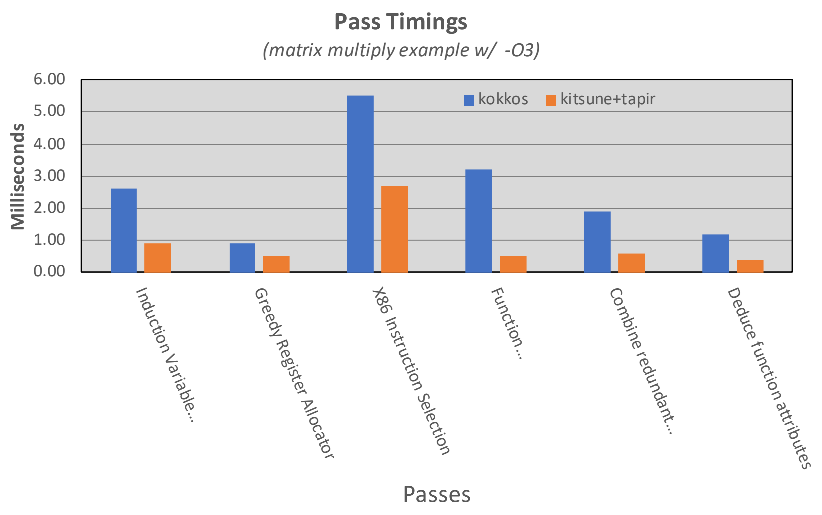
Figure 63: The smaller and more concise intermediate form leads to a smaller number of lines of IR code, smaller executable sizes, and overall faster optimization and compilation times.



(a)



(b)



(c)

In general, the results show that the Tapir IR is roughly 4-times smaller than Clang's LLVM IR for the example benchmarks. What turns out to be significant is the observation that executable sizes are approximately 10-times smaller for Tapir-based code generation. We have yet to fully dissect the full cause for this but suspect part of the issue has to do with template expansion and the way the target runtime system code is generated (note that the results shown reflect the use of the pthreads runtime in Kokkos). Fortunately, this initial size difference does not continue to grow at a similar rate with the introduction of new constructs. However, the initial difference in size does not seem to significantly reduce over time between the two compilation paths nor over separate compilation units.

As mentioned above, the difference in size between the two compiler paths leads directly to longer compilation times that are partially driven by the "short circuit" of C++ code generation with Kitsune and the duration of optimization phases within the middle- and back-end stages of the compiler. Some of the more noticeable differences in the optimization times are highlighted in the bottom portion of Figure 63. While these examples might seem minuscule given their millisecond durations, it is important to remember that this is for a 56-line program and full-sized applications are likely to see significant growth in these costs.

## *Lessons Learned and Future Directions*

Working within the full LLVM toolchain has been a rewarding and challenging effort. We have gained a broader and more significant understanding of some of the complexities our codes face and some of the impact advanced C++ features have on our application codes and their interactions with the supporting compiler. Given that a vast majority of the industry has, or will soon, transition to using the LLVM infrastructure for their production compilers provides a significant opportunity for addressing some of our challenges as well as opportunities for broader collaborations across academia and industry.

These opportunities come with a set of corresponding challenges that have made our experimentation and development activities complex and often met by a significant learning curve. In addition, keeping up with the broader community and the six-month release cycle of LLVM places additional pressure on ever improving feature sets and capabilities. Finally, as the entire LLVM infrastructure (LLVM, Clang, Flang, libc++, lld, lldb, etc.) is approaching well over four-million lines of code. These challenges aside there are numerous activities we are actively working on that we believe will continue to improve the capabilities of both Kitsune and Tapir.

The following two sections highlight two activities that we were unable to fully complete in a stable fashion by the end of the milestone. We continue to actively implement these capabilities so we can take advantage of them within both Kitsune and Tapir.

### GPU BACKEND DEVELOPMENT

One of the substantial advantages of a uniform representation of parallelism is that we can analyze and optimize across hardware boundaries. One place this applies is for GPU development. In a conventional model, there is a hard boundary in code between CPU and GPU: the GPU kernel is compiled by a separate pass, often a separate compiler altogether. This approach prevents any analyses and optimizations from reasoning across these program boundaries. As we move more of our programs to run on GPUs, these boundaries become more and more costly for program performance. An example of such a lost opportunity is loop invariant code motion (LICM). If there is an invariant in code meant to run on a GPU, we would generally prefer to precompute it on the CPU, then share the result across the GPU warps. This is not possible in the conventional setting but comes for free with Kitsune and Tapir.

For these reasons we have developed preliminary GPU targets in Kitsune. The core of the approach is to take canonical Tapir parallel loops and compile them to GPU kernels, inserting the necessary runtime calls automatically. We currently have three targets. For AMD, we compile the body of the parallel loop to AMDGCN, then insert the necessary HIP runtime calls. For Nvidia, we generate PTX and insert Cuda runtime calls. For Intel, we generate SPIRV kernels and insert OpenCL runtime calls. These backends are preliminary, but initial tests confirm that we get optimizations and analyses across CPU-GPU boundaries, as desired. In some cases we are seeing optimizations that provide potential improvements beyond the vendor provided compilers.

### DOMINATOR DAGS

The semantics of LLVM are based on a theory called Single Static Assignment (SSA). The theory uses an always-happens-before relation called a dominance relation to reason about what memory locations can be promoted into immutable values safely. These immutable values are then where all scalar optimizations occur. One limitation of existing theory is that it uses a tree data structure to efficiently approximate the dominance relation. This prevents precise reasoning about dominance, and therefore scalar optimizations, in a number of classes of programs. One such class of programs, an example of which is shown in Figure 64, is where the order of execution of two basic blocks is controlled by a single condition variable. In current LLVM, both the CFG, and therefore the dominator tree, badly approximate the possible behaviors. By extending the dominance relation to be a directed acyclic graph (DAG), using data-sensitive analysis to construct it, we're able to achieve a more precise dominance relation, enabling more optimizations.

Concurrent programs can be seen as a special case of the class of program shown here: the two branches commute, so we can treat the condition variable as non-deterministic and still preserve soundness. The key insight is that fully optimizing parallel codes is infeasible without this improvement on the theory: without it values generated in concurrent blocks will be not to be optimized.

We currently have a preliminary implementation of this extension, extending the existing dominator tree data structure with DAG edges. These are populated by an analysis pass, and used in register promotion. Future work will involve formalizing this approach to ensure confidence in it's correctness, as it touches parts of the compiler that are absolutely crucial. In addition, the asymptotics of the algorithms can likely be improved. Once these two concerns have been addressed, we hope to not only use these changes in Kitsune+Tapir but also upstream the changes to the LLVM project.

```
define f(x){
  entry:
    cond = and x, 1
    br cond, @a@, &b&
  a:
    y = mul 4, x
    call g()
    br cond, @b@, &c&
  b:
    z = add x, x
    call h()
    br cond, @c@, &a&
  c:
    r = add y, z
    ret r
}
```

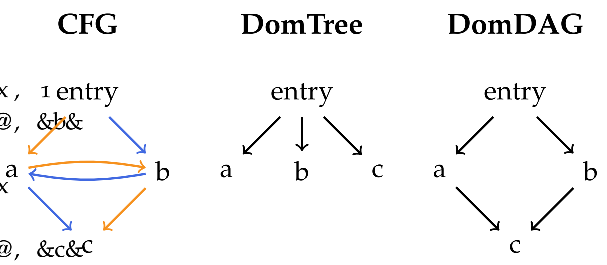


Figure 64: Dominator DAG Example. There are two valid paths through the program, meaning that both `a` and `b` both dominate `c`, something not captured in the current state of the art.

## 3 *The role of* Ristra *at LANL post-ATDM*

LANL has used the opportunity provided by ATDM funding to assess how multi-physics code development could be done differently. Over its 6 year history, the *Ristra* project has had an appreciable technical and cultural impact, and there have been many lessons learned with respect to the implementation of modular capabilities with well-defined interfaces, allowing increasing code sharing and reuse, and an increased separation of concerns between compuational physicists and computer scientists.

As ATDM funding ends, during FY21 LANL is integrating *Ristra* into mainstream ASC funding, with plans in place to sustain and build upon the foundations established under three projects that will fully take off in FY22:

1. ASC CSSE: consolidated software and infrastructure for ASC codes
2. ASC IC: new physics capabilities building on the *Ristra* technologies
3. Next-Generation Platform project: use *FleCSI*-based codes to prototype and co-design advanced architectures and novel hardware together with new physics methods

These topics are the subject of more detailed discussion during the final session of the review.

# Appendix

## A Glossary

***Caliper*** A program instrumentation and performance measurement framework from LLNL

***Capsaicin*** An $S_n$ radiation solver

***CartaBlanca++*** A *FleCSI*-based software environment for prototyping physical models and simulation of a wide variety of physical systems, using an ALE finite-volume method and the Material Point Method (MPM)

***CI*** Continuous integration system: a tool which automatically runs code tests upon code commit or pull request submission

***FleCSALE*** A *FleCSI*-based open-source, unstructured-mesh, single-material, gas dynamics code (direct Eulerian, and Lagrangian plus in-line remap via *Portage)*

***FleCSALE-mm*** An extension of *FleCSALE* to multi-materials with strength

***FleCSI*** The Flexible Computational Science Infrastructure is a compile-time configurable framework designed to support multi-physics application development, and is the key abstraction layer in *Ristra* codes

***FUEL*** A *FLeCSI*-based multi-material ALE hydrodynamics code with SGH (staggered grid) and CCH (cell centered) solvers, targeting low energy density physics

***GitLab*** A web-based DevOps lifecycle tool that provides a Git-repository manager providing wiki, issue-tracking and continuous integration

***HO LO*** A class of multi-scale (scale-bridging) algorithmic acceleration techniques that couple low-order (LO), coarse-grained models with high-order (HO), fine-grained descriptions, in order to resolve macroscopic effects based on microscopic behaviors

***Kitsune*** An LLVM-based parallel-aware compiler (LANL)

***Kokkos*** A programming model in C++ for writing performance portable applications targeting all major HPC platforms

***Legion*** A data-centric task-based parallel programming model for distributed, heterogeneous machines from Stanford, NVIDIA and LANL

***libristra*** A set of support utilities for ristra codes, including mathematical operations (geometry, small matrix operations, and so on), physical units, and input parsing

***LLVM*** A modern, modular compiler toolchain

***Portage*** A modular, extensible framework for general purpose data remapping - between meshes, between particles, or between meshes and particles - in computational physics applications

***Puno*** A $P_1$ unstructured-mesh radiation solver

***Spack*** Spack is a package manager for supercomputers, Linux, and macOS developed by LLNL.

***Symphony*** A *FleCSI*-based radiation hydrodynamics code based on *FleCSALE-mm* and *Puno*, with optional multi-scale (HO LO) radiation transport through coupling to

***Tangram*** A framework for interface reconstruction in computational physics applications that is used by *Portage* for multi-material remap

***Tapir*** Task-based Asymmetric Parallel Intermediate Representation is an extension of LLVM compilers to allow for efficient parallel execution on multicore architectures from MIT; used by *Kitsune*

***tide*** A Lua-based input and setup tool for *Ristra* codes

***Wonton*** A library of wrappers to various unstructured mesh and data (state) managers used by *Portage*

## B ATDM charter

The ATDM sub-program was established towards the end of FY14, with the following goals [NNSA web site, January 2015]:

> The Advanced Technology Development and Mitigation (ATDM) subprogram includes laboratory code and computer engineering and science projects that pursue long-term simulation and computing goals relevant to the broad national security missions of the NNSA. It addresses the need to adapt current integrated design codes and build new codes that are attuned to emerging computing technologies. Performing this work within the scope of the DOE Exascale Computing Initiative (ECI) allows for broader engagement in co-design activities and provides a conduit to HPC vendors to enable next-generation, advanced computing technologies to be of service to the stockpile stewardship mission. Applications developers, along with computational and computer scientists, are to build a computational infrastructure and develop a new generation of weapon design codes that will efficiently utilize the hardware capabilities anticipated in exascale-class systems. As part of this subprogram's work scope, the ASC Program has engaged with the DOE Office of Advanced Scientific Computing Research to address the barriers to exascale and evolving architectures.

Further, ATDM called for each laboratory to establish a Next-Generation Code Development and Applications (ATDM CDA) project that

> ... is focused on developing new simulation tools that address emerging HPC challenges of massive, heterogeneous parallelism using novel programming models and data management. Modern codes will be developed through co-design of applications by laboratory scientists and engineers and co-design of computer systems by computer vendors. The end product of this work is a next-generation set of simulation tools that may complement and/or replace the current set of production tools for the Nuclear Security Enterprise (NSE).

## C Advanced Technology Development and Mitigation (ATDM) Target (Level 1 Milestone Definition)

The ATDM subprogram was created to mitigate the risks of application code portability and performance on future HPC architectures. ATDM includes two basic elements: development of new codes that are more architecture agnostic than existing codes, and the underlying hardware and software issues associated with future architectures. Full implementation, verification, and validation of the new codes will likely take a decade or more; however, five years is a sufficient length of time to make substantial progress. This milestone focuses on the new codes and is meant to take stock of the ongoing progress, document lessons learned, and identify programmatic investments needed for the next phase of development.

**The results of this ATDM L1 milestone shall reflect how each team established their baseline of performance and capture an initial set of acceptance criteria for end users.**

**Action**

Each laboratory shall do the following:

1. Perform a series of NW relevant calculations in 3D using up to and including at least 25% Sierra

and either 50% Trinity or 100% Astra.

2. Perform a calculation of the same scenario including the same physics but with a current production code on one of the chosen platforms at one of the chosen scales.

3. Develop a set of metrics for evaluation based on the criteria described below.

4. Assess both the new codes and existing codes (modulo scaling capability) with those metrics.

5. Compare the results.

**Assessment**

The labs shall develop qualitative and quantitative metrics and assess the ATDM and existing codes on the basis of the following criteria:

1. **Mission Impact:** Define a metric(s) that will express the status of both new and existing codes in terms relevant to end users. Areas to consider in the assessment include: problem set up times, code robustness, turn-around time, storage requirements, post-processing capabilities, visualization capabilities, degree of verification and validation, solution accuracy, workflow, etc.

2. **Portability:** Evaluate code portability by running the same application source code on different platforms and assessing the results. Identify the relative amount of source code that is platform agnostic vice the source code that is unique to a specific platform.

3. **Developer Productivity:** Document and describe the ease of improving, modifying, or extending the codes, and training of staff. Some points to consider include: relative ease of implementing new models/methods/algorithms into the codes; improvements in software quality; maintenance and reuse of code in existing production capabilities; knowledge transfer, education, and training of developers.

4. **Code Performance:** Define and describe code capability in terms of computational performance. Measures to consider include: time-to-solution, percentage of peak FLOPs, strong and weak scaling, efficiency using the computational platform, etc.

**Reporting Requirements**

Document the results of the assessment, discuss lessons learned, and propose changes to address performance gaps for the next 5 years starting in FY 2021. Possible topics to consider include:

1. What has been accomplished up to the completion of this milestone? What were the significant innovations? What new science and/or mission impact has ATDM made possible? Have the new achieved advances in terms of the assessment relative to existing codes. The latter may be communicated to NNSA independently.

2. What work remains to for the codes to reach complete production status? What are the requirements and plans to meet these requirements to achieve this status? How will this work be carried into the planning cycle for IC or for ASC writ large?

3. Are there any high risk/high reward efforts that failed? If so, are there potential mitigation strategies that should be considered moving forward?

## D Project timeline (selected highlights)

FY14

- ATDM sub-program created
- Task force established at LANL to provide recommendations for the new code effort
- Project leadership and high-level goals established
- Initial plans presented at HQ

*FY15*

- Project kickoff, initial plan for the first FY to focus on prototyping and co-design discussions and experiments
- Initial software development processes established, initially based on Atlassian tools with Jenkins for CI (continuous integration) testing
- Automated mirroring of *Ristra* repositories between LANL networks
- Requirements gathering based on foundational physics V&V principles established through discussions with subject matter experts
- Assessment of the state-of-the-art of variety of mesh-based and mesh-free methods
- Decision to focus on unstructured ALE methods, and subsequent emergence of sub-teams to focus future development:
  - Integration across the project
  - *FleCSI* developing foundational compile-time-configurable multi-physics infrastructure
  - *Portage* developing a stand-alone remap and link capability
  - ALE team focusing on modern hydrodynamic methods on general polyhedral meshes
  - Scalable methods focusing on new multi-scale methods and multi-physics couplings beyond the traditional operator split

*FY16*

- Integration
  - ATDM falls under aegis of Exascale Computing Project: presentations to national ECP leadership, and LANL Director McMillan
  - Hosted Git and Gitlab training for a broad LANL audience
- *FleCSI*
  - Data and execution model design and code review; initial unstructured mesh and particle topologies; initial sparse data layout support
  - Initial *FleCSI* backends for Legion and MPI
  - Open source licensed
  - First co-design iteration of dynamic polymorphism for multi-physics applications in *FleCSI*
  -

  *Portage*
  - Adoption of existing *Jali* mesh library for protyping phase. *Jali* enhanced with a tile capability to support multi-material and multi-scale studies.
  - Initial 3D capability with 1st and 2nd order remap schemes
  - Early explorations on GPUs
  - Initial distributed memory mode using third-party *DTK* library for parallel search
  - Support for *FleCSI* unstructured mesh format
- ALE
  - Proxy application *FleCSALE* solves the 2D Euler equations on a fixed mesh, and using the cell-centered method of Maire on a Lagrangian mesh
  - Existing *Cercion* multi-physics proxy used for multi-material data structure requirements
- Scalable methods
  - *Puno* low-order (P1) radiation model developed for a multi-scale prototype that will use Sn for the higher-order method. Building on HOLO work in the *CoCoMANS* LDRD project and lessons learned integrating that capability in *xRAGE*, the

  code uses the *Jali* mesh model and *Trillinos* to explore structural challenges for integrating multi-scale methods and 3$^{rd}$ party libraries.
  - Exploration time-explicit radiation transport methods

- First ASC ATDM Level 2 milestone passed

## *FY17*

- Integration
  - Established plans for problem setup (workflow), visualization and data analysis, and I/O
  - Cross-project multi-material data structure co-design
  - *libristra* cross-project utility library established
- *FleCSI*
  - *FleCSI/Legion* integration starts in earnest in collaboration with *Legion* developers at Stanford and NVIDIA
  - Initial *FleCSI/Legion* demonstration of a hydrodynamics method; performance improvements for *FleCSI/MPI* and *FleCSI/Legion*
  - Initial support for sparse multi-material data structures
  - *In situ* visualization using *Paraview/Catalyst* demonstrated in *FleCSALE*
- *Portage*
  - Open source license
  - *Tangram* interface-reconstruction, and *Wonton* mesh wrapper libraries intiated
  - Mesh-mesh remap through an intermediate particle representation
- ALE
  - *FleCSALE* refactor implemented to include stress tensor
  - Export-controlled *FleCSALE-mm* code forked supporting multi-material hydrodynamics with strength. *FUEL* code forked to support advanced material model research.
  - Five-material cylindrical implosion demonstrated in 2D RZ Lagrangian, with sparse multi-physics model support and 3D Lagrangian with a dense multi-physics model support
  - 3D Taylor-Anvil impact experiment simulation

  - Strength models in *FleCSALE-mm:* Hyper- and hypo-elastic models; investigations of VPSC (Visco-Plastic Self-Consistent) grain-structure-aware strength model

- Scalable methods

  - Investigation of task parallel methods for exploration of explicit radiation flow schemes with simple TN burn, and alternative operator split algorithms in ICF problems
  - Improved the LO part of the $S_N$-HO LO scheme for radiation hydrodynamics by adopting a Discontinuous Galerkin in place of a Finite Volume. Adopted by EAP and *Ristra* projects

- Second ASC ATDM Level 2 milestone passed

### *FY18*

- Integration

  - Open source license for *libristra*
  - Rad-hydro code design iterations including members across the project
  - *Symphony* radiation hydrodynamics code initiated through coupling of *FleCSALE-mm* and *Puno* P1 radiation. HO LO capability demonstrated by coupling with 3rd party *Capsaicin* Sn library for the HO solve.

- *FleCSI*

  - Scaling limitations in current *Legion* implementation addressed in the short term, with plans for long-term "control replication" solution discussed with *Legion* developers
  - Co-design of sparse/ragged data layouts and MPI interoperability within *Symphony* multi-physics code
  - Runtime selectable physics methods enabled through *FleCSI*
  - Initial discussions for integration of Legion control replication into *FleCSI*

- *Portage*

  - Open source licenses for *Tangram* and *Wonton*
  - Suite of challenging remap validation tests developed
  - 2D and 3D VOF and MOF interface reconstruction demonstrated in *Tangram*; 2D and 3D multi-material remap demonstrated with *Tangram* called from *Portage*

  - Performance improvements in intersection algorithms

- ALE
  - FleCSALE-mm supports Sesame EOS tables
  - Advanced material models
    * A material point method (MPM) algorithm implemented through a particle/mesh topology prototype in *FleCSALE-mm*
    * VPSC strength model interfaced to *FUEL*; targeting a CUDA rewrite of VPSC for a GPU multi-scale demonstration problem

- Scalable methods
  - *Puno* P1 radiation code refactored on *FleCSI* unstructured mesh
  - Scalable methods team folded into Integration effort for FY19

- Third ASC ATDM Level 2 milestone passed

## *FY19*

- Integration
  - *Symphony: Puno* solver updated to use improved lumped discontinuous Galerkin method
  - Initial coupled (self-consistent IMEX) rad hydro method added to *Symphony*
  - Simple reactive burn model in *Symphony*
  - Integration of HED and LED capabilities (rad hydro + multimaterial strength)
  - Checkpoint/restart in *Symphony*
  - Consolidating on *Spack* for managing *Ristra*'s complex and diverse build environment
  - Migration of many SW repositories to Gitlab for improved CI testing

- *FleCSI*
  - New sparse/ragged data layouts integrated into *FleCSI* Initial plan outlined for *Kokkos* integration into *FleCSI*, and demonstrated for *OpenMP* and *CUDA* targets
  - Simple meshing tools, for mesh generation and parallel mesh I/O
  - Integration of LLNL *Caliper* performance counters

  - Major refactor ("*FleCSI 2.0*") started, based on lessons learned, feedback from developers

- *Portage*

  - *Portage 2.0* released
  - Multi-material validation tests for *Portage/Tangram*
  - Improved *FleCSI* support in *Portage*

- ALE

  - First *Portage*-based remap capability in *FleCSALE-mm*
  - Reference ALE capability in *FUEL*
  - *Legion* + VPSC: improved CUDA VPSC implementation
  - *FleCSALE-mm* demonstrated on *Sierra*-architecture nodes with *MPI* and *Legion* backends

- Fourth ASC ATDM Level 2 milestone passed

*FY20*

- Integration

  - Consolidation of FY19 codes as the basis for milestone demonstrations Multi-group diffusion radiation solver in *Puno* and *Symphony*
  - Multi-material ALE using *Portage/Tangram* for remap demonstrated in *Symphony*
  - All additional required physics integrated and demonstrated in an integral problem
  - *TIDE,* a *Lua*-based, compile-time/run-time problem setup tool developed
  - *Spack*-generated modules for *Ristra* development deployed on production platforms
  - *Crosslink/ParMesh* parametric scalable meshing tool targeted for *Symphony*
  - Build system improvements with better integration of *Paraview/-Catalyst, Kokkos, Legion*
  - Integral tests included in CI testing
  - Optimization of performance and scaling in *Symphony*

- *FleCSI*

  - Identified stable *FleCSI 1.4* branch as target for milestone demonstrations
  - *FleCSI 2.0* development continues
  - *N* to *M HDF5*-based checkpoint/restart prototypes demonstrated for *MPI and Legion*
  - *FleCSPH* smoothed particle hydrodynamics code used to prototype FleCSI tree topology
  - *CartaBlanca++*, a FleCSI-based code featuring the MPM (Material Point Method) demonstrated in additive manufacturing and fragmentation field applications
  - *FleCSI/Kokkos* improvements, in collaboration with *Kokkos* team at SNL
  - Memory and scaling issues in *FleCSI/Legion*: still awaiting resolution via *Legion* refactor
  - *Kitsune FleCSI*-aware parallel compiler successfully compiling and running *FleCSALE-mm* on NVIDIA GPUS

- *Portage*
  - Part-by-part remapping
  - Example *FleCSI*-based driver for *Portage;* New *Tangram* release
  - Co-design of improved interface for inline remap in *FleCSI*-based codes
  - Refactor of build system based on modern *CMake* best practices

- ALE
  - Multi-material ALE using *Portage* for inline remap demonstrated in *Symphony*
  - Improved *VPSC* model for robustness in high-deformation applications
  - Demonstrations of multi-scale methods using advanced hydrodynamic algorithms and high-fidelity material models

- ASC ATDM Level 1 milestone mid-cycle review
- ASC ATDM Level 1 milestone postponed due to covid-19 pandemic

**FY21**

- Final preparation for ASC ATDM Level 1 milestone, with large-scale physics demonstrations on *Trinity, Sierra,* and *Astra*
- *FleCSI 2.0* released
- *Portage 3.0* released

## *References*

[1] Andrew Barlow, Pierre-Henri Maire, William Rider, Robert Rieben, and Mikhail Shashkov. Arbitrary lagrangian-eulerian methods for modeling high-speed compressible multimaterial flows. *Journal of Computational Physics*, 322(c):603—665, 2016. doi: 10.1016/j.jcp.2016.07.001.

[2] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[3] J. F. Bingert, R. M. Suter, J. Lind, S. F. Li, R. Pokharel, and C. P. Trujillo. High-Energy Diffraction Microscopy Characterization of Spall Damage. In *Dynamic Behavior of Materials, Volume 1*, pages 397–403. Springer International Publishing, 2014.

[4] S. Chen, G. Gray III, and S. Bingert. Mechanical properties and constitutive relations for tantalum and tantalum alloys under high-rate deformation. Technical Report No. LA-UR-96-0602, CONF-960202-24, Los Alamos National Laboratory, 1996.

[5] V. Chiravalle and N. Morgan. A 3D finite element ALE method using an approximate Riemann solution. *International Journal for Numerical Methods in Fluids*, 83:642–663, 2016.

[6] Clang. Clang: a C language family frontend for LLVM. http://clang.llvm.org, November 2020.

[7] S. S. Dhinwal, L. S. Toth, R. Lapovok, and P. D. Hodgson. Tailoring one-pass asymmetric rolling of extra low carbon steel for shear texture and recrystallization. *Materials*, 12 (12), 2019. ISSN 1996-1944. doi: 10.3390/ma12121935. URL https://www.mdpi.com/1996-1944/12/12/1935.

[8] Gary Dilts. Estimation of integral operators on random data. Technical Report LA-UR-17-23408, Los Alamos National Laboratory, 2017.

[9] H. Lim, J. D. Carroll, C. C. Battaile, S. R. Chen, A. P. Moore, and J. M. D. Lane. Anisotropy and strain localization in dynamic impact experiments of tantalum single crystals. *Scientific Reports*, 8(1):5540, 2018. doi: 10.1038/s41598-018-23879-1. URL https://doi.org/10.1038/s41598-018-23879-1.

[10] LLVM. The LLVM Compiler Infrastructure. http://www.llvm.org, November 2020.

[11] P. J. Maudlin, J. F. Bingert, J. W. House, and S. R. Chen. On the modeling of the taylor cylinder impact test for orthotropic textured materials: experiments and simulations. *International Journal of Plasticity*, 15(2):139 – 166, 1999. ISSN 0749-6419. doi: https://doi.org/10.1016/S0749-6419(98)00058-8. URL http://www.sciencedirect.com/science/article/pii/S0749641998000588.

[12] R. T. Olson, E. K. Cerreta, C. Morris, A. M. Montoya, F. G. Mariam, A. Saunders, R. S. King, E. N. Brown, G. T. Gray, and J. F. Bingert. The effect of microstructure on Rayleigh-Taylor instability growth in solids. *Journal of Physics: Conference Series*, 500 (11):112048, May 2014. doi: 10.1088/1742-6596/500/11/112048. URL https://doi.org/10.1088%2F1742-6596%2F500%2F11%2F112048.

[13] C. N. Reid. *Deformation Geometry for Materials Scientists*. International Series on Materials Science and Technology. Pergamon, 1973.

[14] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Progr amming*, PPoPP '17, page 249–265, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344937. doi: 10.1145/3018743.3018758. URL https://doi.org/10.1145/3018743.3018758.

[15] George Stelle, William S. Moses, Stephen L. Olivier, and Patrick McCormick. Openmpir: Implementing openmp tasks with tapir. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM-HPC'17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450355650. doi: 10.1145/3148173.3148186. URL https://doi.org/10.1145/3148173.3148186.

[16] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[17] Steven Zalesak. Fully multidimensional flux-corrected transport algorithms for fluids. *Journal of Computational Physics*, 31(3): 335–362, 1979. doi: 10.1016/0021-9991(79)90051-2.

[18] M. Zecevic and M. Knezevic. A new visco-plastic self-consistent formulation implicit in dislocation-based hardening within implicit finite elements: application to high strain rate and impact deformation of tantalum. *Computer Methods in Applied Mechanics and Engineering*, 341:888–916, 2018.