A software architecture for future multi-physics applications

David Daniel
Ristra project co-lead

Salishan Conference on High-Speed Computing, Gleneden Beach, Oregon
April 24, 2019

LA-UR-19-23560

Ristra, a next-generation code project at Los Alamos, will create a set of codes that solve ASC-relevant multi-physics problems using modern computational methods suitable for emerging extreme-scale computer architectures.

Acknowledgements

The Ristra team – many contributors across diverse disciplines

Aimee Hungerford (Physics Lead)

David Daniel (CS Lead)

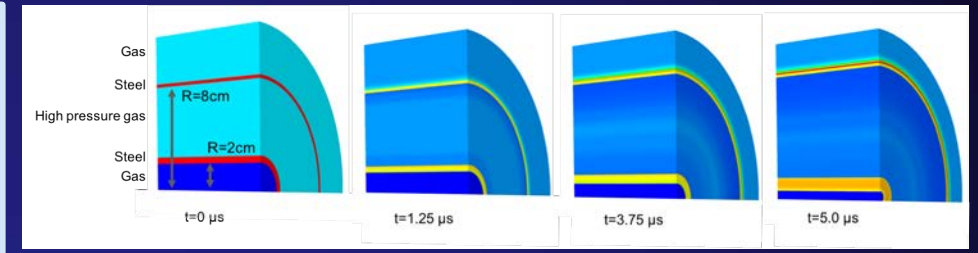
Matt Bement, Ben Bergen, Marc Charest, Vincent Chiravalle, Irina Demeshko, Gary Dilts, Josh Dolence, Charles Ferenbaugh, Chris Fryer, Rao Garimella, Pascal Grosset, Scot Halverson, Angela Herring, Brian Jean, Christoph Junghans, Eugene Kikinzon, Ricardo Lebensohn, Ollie Lo, Chris Malone, Len Margolin, Pat McCormick, Nathaniel Morgan, Ryosuke Park, Robert Pavel, Navamita Ray, Jonathan Regele, Mike Rogers, Misha Shashkov, Danny Shevitz, Martin Staley, Galen Shipman, Karen Tsai, Jan Velechovsky, Daniele Versino, … & others.

The Los Alamos Ristra project (ECP and ASC)

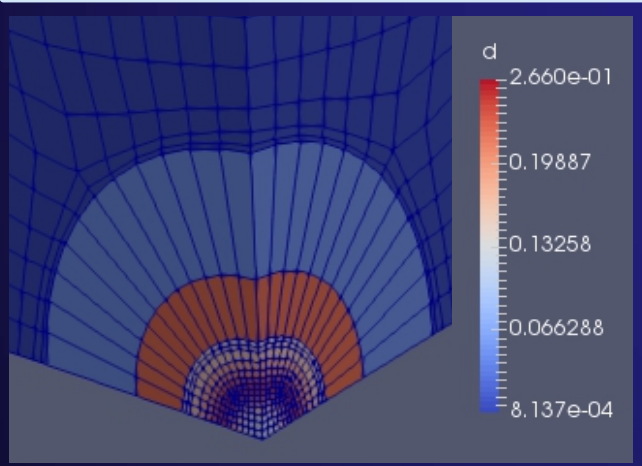
Next-generation multi-physics for basic science and national security applications

Ever changing scientific questions & Rapidly changing HPC hardware

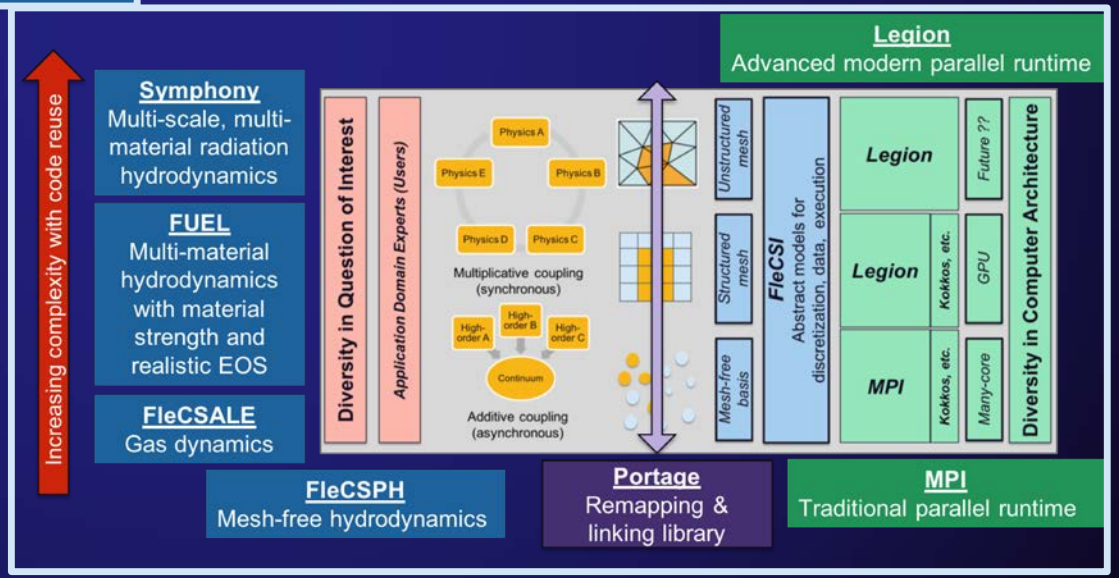
- Ristra is creating a ***flexible and agile*** toolkit for next-generation multi-physics
- FleCSI abstraction layer separates concerns of physics and CS developers
- Portage remap and link library connects meshes, packages, and codes



Ristra *FUEL* code results for a multi-material hydrodynamic test problem with contact discontinuities, shocks, and release waves.



Ristra *Symphony* code results for an imploding ICF capsule (N12032)



Heterogeneity at many levels in multi-physics codes

- Hardware
  - The primary focus of this meeting
  - Here today, but will ebb and flow over time
- Physics, applied mathematics, and CS
  - **Static** choice of relevant physics and appropriate methods at a given simulated time
  - **Dynamic** evolution of relevant physics and methods with simulated time
  - **Dynamic** evolution of relevant physics and methods over the lifetime of a code as the field progresses and customer demands change

How should developers of new multi-physics simulation codes prepare to meet these challenges today, and respond to their evolution over time?

How should developers of new multi-physics simulation codes prepare to meet the challenges of heterogeneity today, and respond to their evolution over time?

Our approach is to look for high-level principles that reveal

- Parallelism
  - Favor data parallelism friendly to hardware
- Asynchrony
  - Latency hiding of interaction between heterogenous components
- Hierarchy
  - Physics and algorithms, for scalability

Looking for parallelism from first principles

Physics is data parallel!

- Physics is fundamentally *local* and *causal*
  - *Scalable data parallelism*
- However, direct (explicit) methods may be infeasible (too expensive)
- Most physics simulations rely on quasi-static assumptions (e.g. finite elements
  - Gain efficiency, lose scalability

		Concurrency *Property of the program*	
Parallelism *Property of the machine*		***Sequential***	***Concurrent***
	Serial	Traditional program	Multi-tasking program
	Parallel	Deterministic parallelism	General parallelism

Physics naturally lies here (explicit methods)

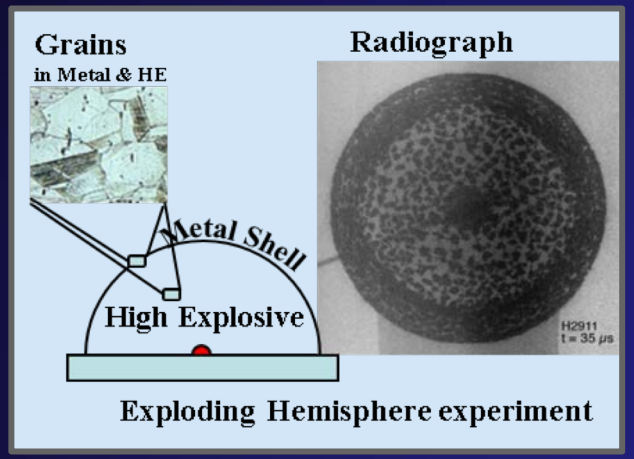
The choice of algorithm may push computational physics here at a scalability cost

For physics beyond exascale, reconsider algorithmic tradeoffs

Table after G. Blelloch

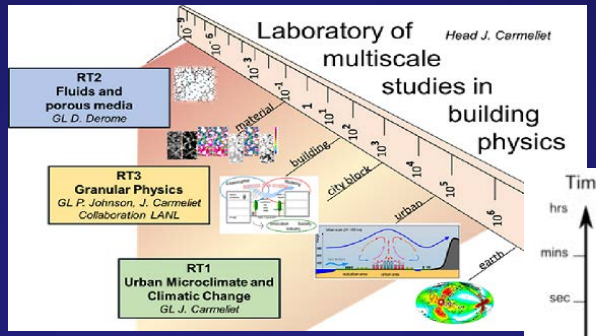
Looking for hierarchy and asynchrony

– Multi-scale methods provide a path to higher fidelity physics



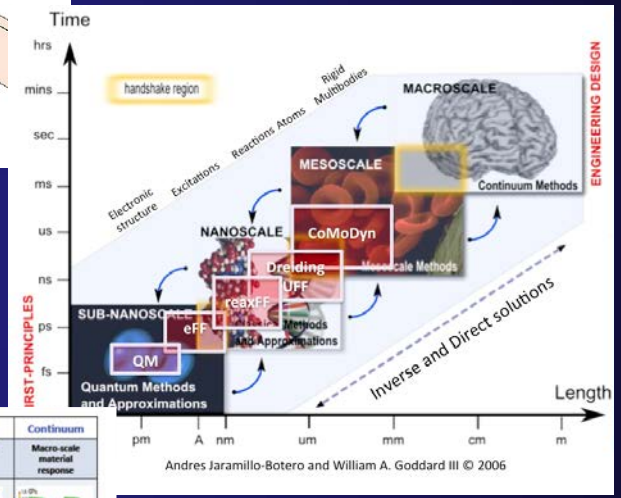
Resolving grain-level physics: improved fidelity in experiment *and* simulation

- **Models at different scales (*fine* to *coarse*) & bridging between them (*multi-scale methods*)**
- **Coarse: multi-physics coupling**
- **Fine: higher fidelity *and* asynchronous concurrency**



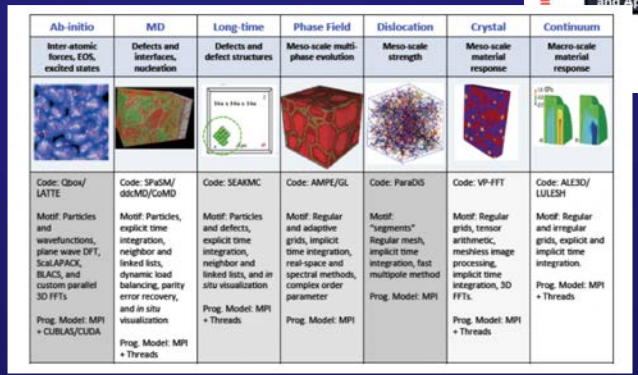
Climate

*http://empa.ch



Biology

*http://wag.caltech.edu



Ab-initio	MD	Long-time	Phase Field	Dislocation	Crystal	Continuum
Inter-atomic forces, EOS, excited states	Defects and interfaces, nucleation	Defects and defect structures	Meso-scale multi-phase evolution	Meso-scale strength	Meso-scale material response	Macro-scale material response
Code: Qbox/ LATTE Motif: Particles and wavefunctions, plane wave DFT, ScaLAPACK, BLACS, and custom parallel 3D FFTs Prog. Model: MPI + CUBLAS/CUDA	Code: SPaSM/ ddcMD/CoMD Motif: Particles, explicit time integration, neighbor and linked lists, dynamic load balancing, parity error recovery, and *in situ* visualization Prog. Model: MPI + Threads	Code: SEAKMC Motif: Particles and defects, explicit time integration, neighbor and linked lists, and *in situ* visualization Prog. Model: MPI + Threads	Code: AMPE/GL Motif: Regular and adaptive grids, implicit time integration, real-space and spectral methods, complex order parameter Prog. Model: MPI	Code: ParaDIS Motif: "segments" Regular mesh, implicit time integration, fast multipole method Prog. Model: MPI	Code: VP-FFT Motif: Regular grids, tensor arithmetic, meshless image processing, implicit time integration, 3D FFTs. Prog. Model: MPI + Threads	Code: ALE3D/ LULESH Motif: Regular and irregular grids, explicit and implicit time integration. Prog. Model: MPI + Threads

Materials

*Ex Mat Ex

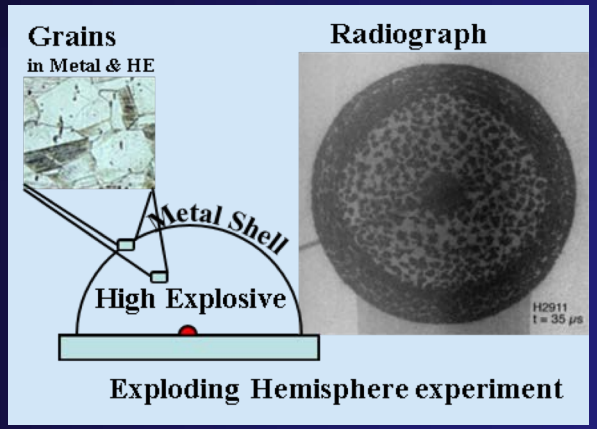
Ristra: Next-generation codes for exascale challenges

– Balancing Future physics; Future methods; Future Computers

– Agile response to new questions; e.g., advanced material modeling

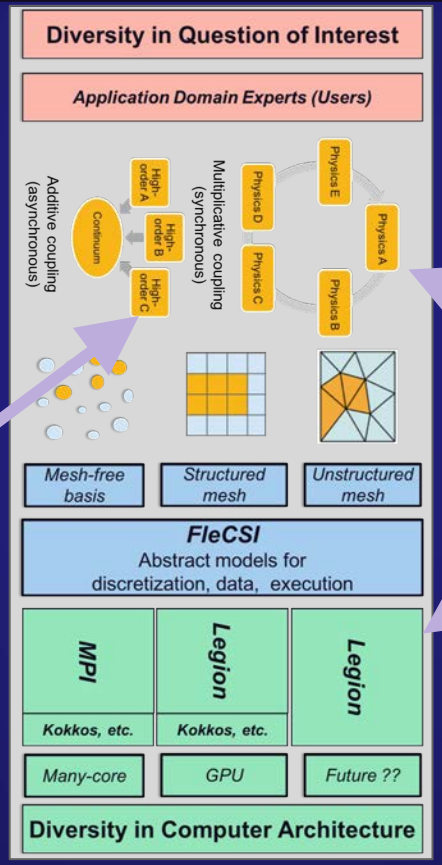
Multi-scale methods

Novel programming models



Resolving grain-level physics: improved fidelity in experiment (DARHT, MaRIE) *and* simulation

- **Models at different scales (*fine* to *coarse*) & bridging between them (*multi-scale methods*)**
- **<u>Coarse</u>: multi-physics coupling**
- **<u>Fine</u>: higher fidelity *<u>and</u>* asynchronous concurrency**



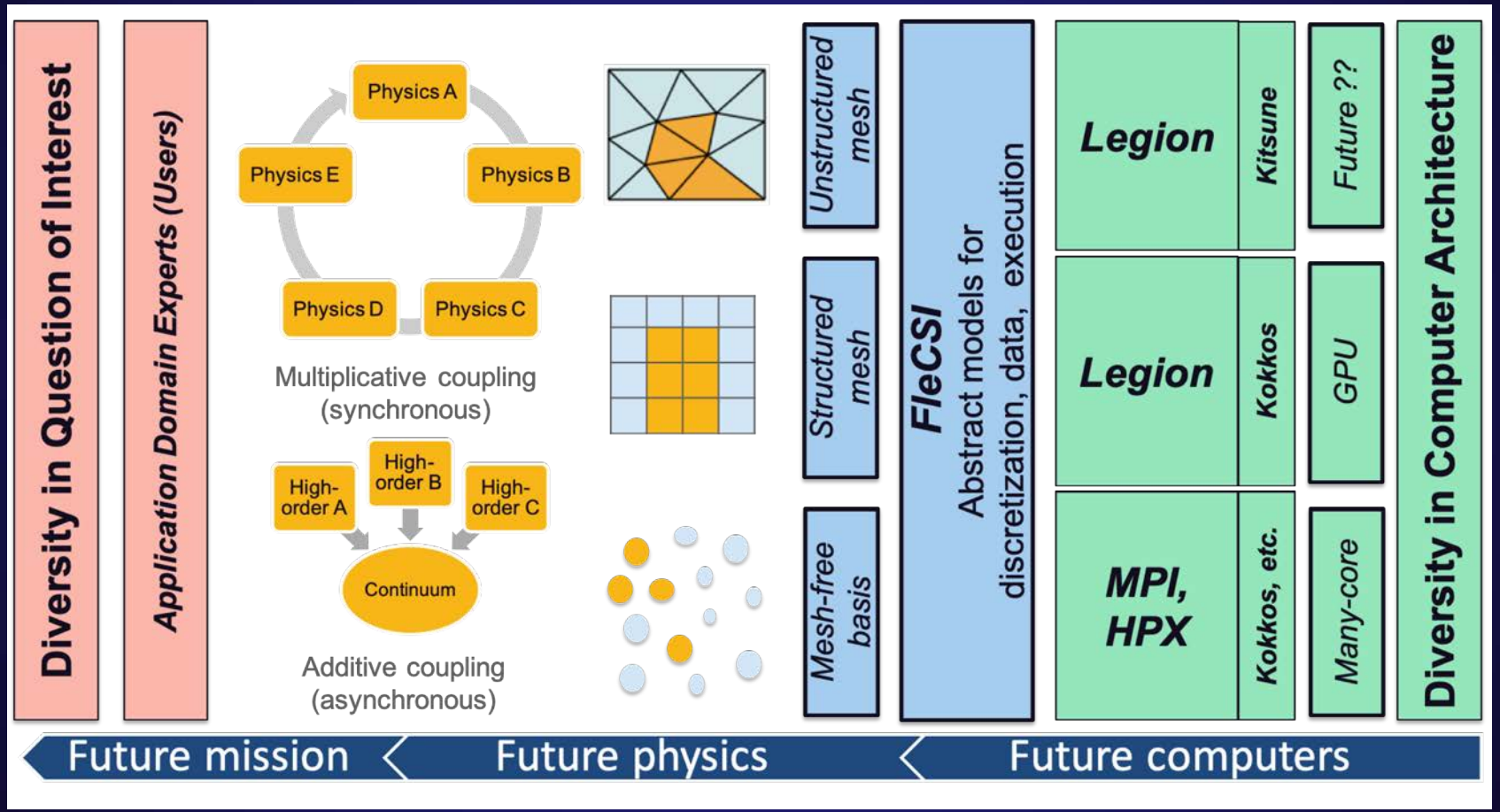
Common theme at exascale: need for <u>asynchronous</u> methods tolerant of latency variability within a computational node, and across an extreme-scale system

- **Traditional physics and CS methods (operator split, MPI) have poor asynchrony**
- **New programming models expose more parallelism for asynchronous execution**

Building leadership in computational science from advanced materials to novel programming models

Ristra SW architecture targets flexibility in a volatile future

– FleCSI separates concerns of physics and CS development



Ristra: Status of applications and components

– *Variety of application codes have been developed using FleCSI*
– *Two different parallel runtimes have been fielded, with more to come*

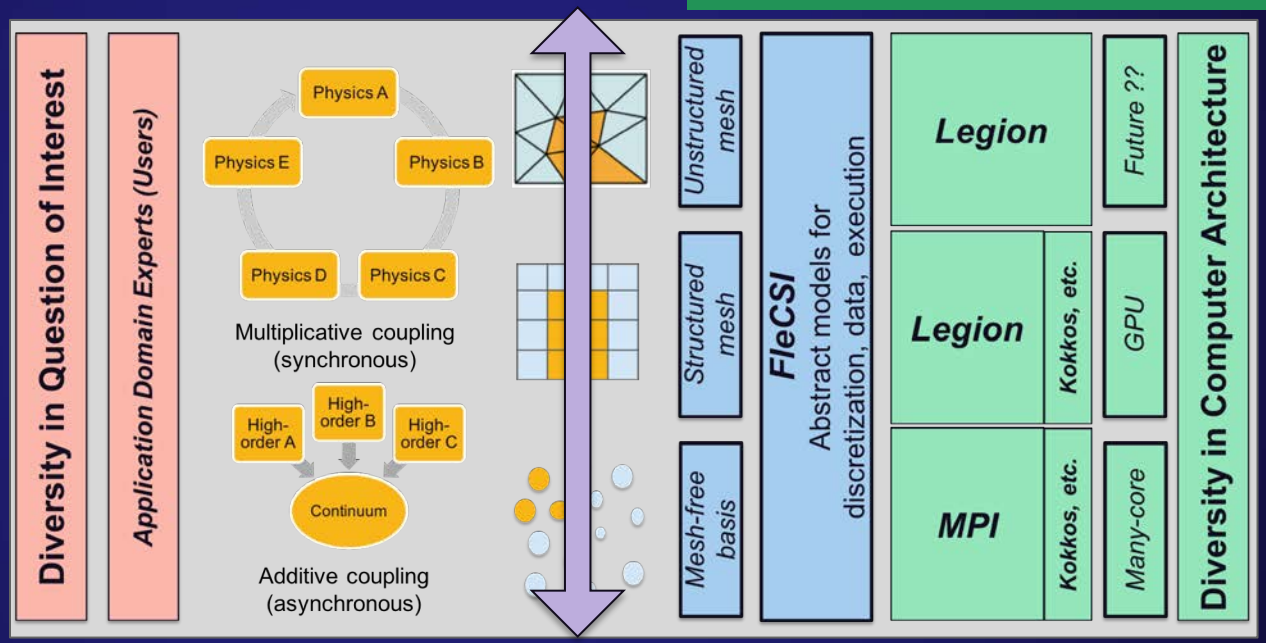
Legion
Advanced modern parallel runtime

Increasing complexity with code reuse

Symphony
Multi-scale, multi-material radiation hydrodynamics

FUEL
Multi-material hydrodynamics with material strength and realistic EOS

FleCSALE
Gas dynamics



FleCSPH
Mesh-free hydrodynamics

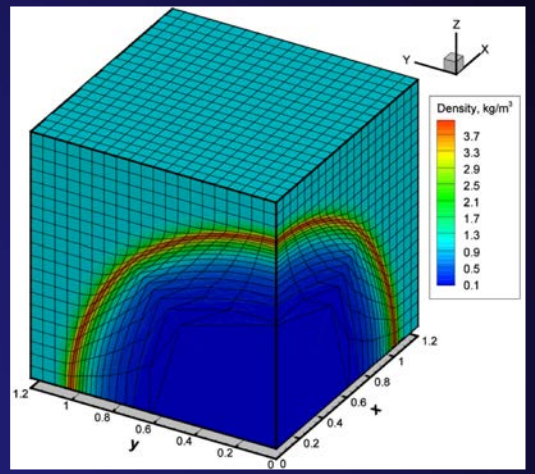
Portage
Remapping & linking library

MPI
Traditional parallel runtime

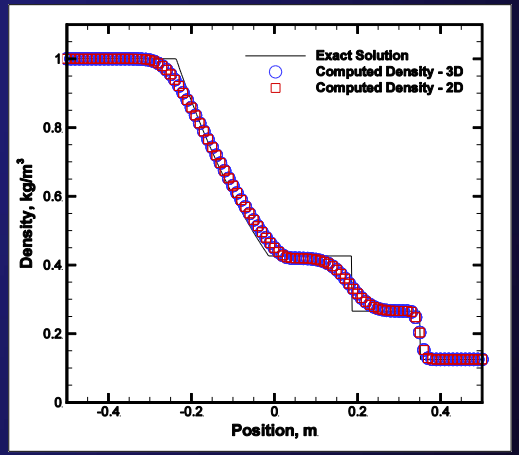
What is FleCSI?

FleCSI is a C++ programming system for developing multi-physics simulation codes

- Runtime abstraction layer
  - High-level user interface, mid-level static specialization, low-level building blocks, tasking and fine-grained threading back-ends
- Programming model
  - Data model
  - Execution model
  - Control models
- Useful data structure support
  - Mesh topology
  - N-Tree (N=3 → Octree) topology
  - Set topology



FleCSI: Pure 3D Lagrangian Sedov



FleCSI: 2D/3D Eulerian Sod

FleCSI Core Data Structures

`flecsi::topology::mesh_topology__`

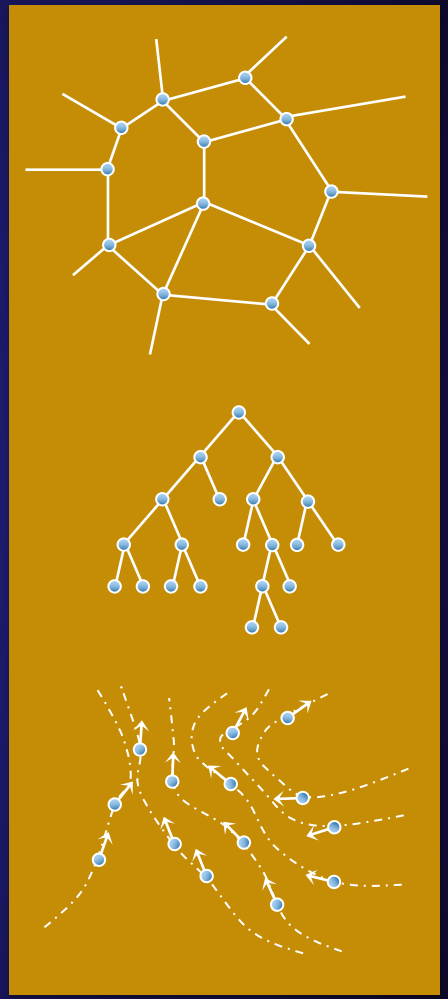
- Support for unstructured meshes with user-defined mesh entity types, and user-defined adjacency storage

`flecsi::topology::tree_topology__`

- Support for hashed trees with user-defined node types, and user-defined relational functions, e.g., “who are my neighbors?”

`flecsi::topology::set_topology__`

- Support for sets of user-defined entities, e.g., non-interacting particles, and user-defined rules for entity migration, coloring, and binning



FleCSI Core Data Structures

`flecsi::topology::mesh_topology__`

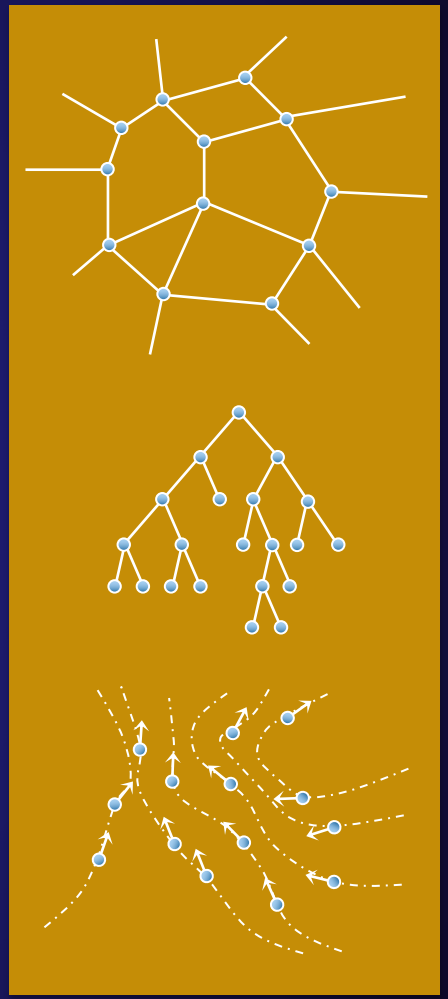
- Hydrodynamics (Eulerian, Lagrangian, ALE, Re-ALE, DG), Radiation/Heat Conductivity

`flecsi::topology::tree_topology__`

- N-Body, Smoothed-Particle Hydrodynamics

`flecsi::topology::set_topology__`

- Particle-in-Cell (PIC), Material-Point Method (MPM), Charged/Neutral Particle Transport



FleCSI programming

In one slide!

- Application-specific topology type is derived from core FleCSI topologies through compile-time policies
- Data model of “fields” over “index spaces” is naturally data parallel
- Storage class/policy for fields (policy) can be dense or sparse
- Task-based execution model

```
flecsi_register_field(mesh_t, example, field, double, sparse, 1, mesh);

…



void ns::init(mesh<ro> mesh, sparse_field_mutator f) {

  for (auto c: mesh.cells()) {
    const size_t random = (rand()/double{RAND_MAX}) * 5;

    for (size_t i{0}; i < random; ++i) {
      const size_t entry = (rand()/double{RAND_MAX}) * 5;
      f(c, entry) = entry;
    }
  }
}
flecsi_register_task(init, ns, loc, single);

…

void driver() {
  auto m = flecsi_get_client_handle(mesh_t, clients, mesh);
  auto f = flecsi_get_mutator(m, example, field, double, sparse, 0, 5);
  flecsi_execute_task(init, ns, single, m, f);
}
```

FleCSI programming

In one slide!

- Application-spec
  topology type is c
  from core FleCSI
  topologies through
  compile-time policies
- Data model of “fields”
  over “index spaces” is
  naturally data parallel
- Storage class/policy for
  fields (policy) can be
  dense or sparse
- Task-based execution
  model

With annotations ☹

```
flecsi_register_field(mesh_t, example, field, double, sparse, 1, mesh);

void ns::init(mesh<ro> mesh, sparse_field_mutator f) {

  for (auto c: mesh.cells()) {
    const size_t random = (rand()/double{RAND_MAX}) * 5;

    for (size_t i{0}; i < random; ++i) {
      const size_t entry = (rand()/double{RAND_MAX}) * 5;
      f(c, entry) = entry;
    }
  }
}
flecsi_register_task(init, ns, loc, single);

…

void driver() {
  auto m = flecsi_get_client_handle(mesh_t, clients, mesh);
  auto f = flecsi_get_mutator(m, example, field, double, sparse, 0, 5);
  flecsi_execute_task(init, ns, single, m, f);
}
```

FleCSI programming

In one slide!

- Application-specific topology type is derived from core FleCSI topologies through compile-time policies
- Data model of “fields” over “index spaces” is naturally data parallel
- Storage class/policy for fields (policy) can be dense or sparse
- Task-based execution model

```
flecsi_register_field(mesh_t, example, field, double, sparse, 1, mesh);
```

application-specific topology type

index space

```
…

void ns::init(mesh<ro> mesh, sparse_field_mutator f) {

  for (auto c: mesh.cells()) {
    const size_t random = (rand()/double{RAND_MAX}) * 5;

    for (size_t i{0}; i < random; ++i) {
      const size_t entry = (rand()/double{RAND_MAX}) * 5;
      f(c, entry) = entry;
    }
  }
}
flecsi_register_task(init, ns, loc, single);

…

void driver() {
  auto m = flecsi_get_client_handle(mesh_t, clients, mesh);
  auto f = flecsi_get_mutator(m, example, field, double, sparse, 0, 5);
  flecsi_execute_task(init, ns, single, m, f);
}
```

FleCSI programming

In one slide!

- Application-specific topology type is derived from core FleCSI topologies through compile-time policies
- Data model of “fields” over “index spaces” is naturally data parallel
- Storage class/policy for fields (policy) can be dense or sparse
- Task-based execution model

type *number of versions*

```
flecsi_register_field(mesh_t, example, field, double, sparse, 1, mesh);
```

field name *index space*

```
…

void ns::init(mesh<ro> mesh, sparse_field_mutator f) {

  for (auto c: mesh.cells()) {
    const size_t random = (rand()/double{RAND_MAX}) * 5;

    for (size_t i{0}; i < random; ++i) {
      const size_t entry = (rand()/double{RAND_MAX}) * 5;
      f(c, entry) = entry;
    }
  }
}
flecsi_register_task(init, ns, loc, single);

…

void driver() {
  auto m = flecsi_get_client_handle(mesh_t, clients, mesh);
  auto f = flecsi_get_mutator(m, example, field, double, sparse, 0, 5);
  flecsi_execute_task(init, ns, single, m, f);
}
```

FleCSI programming

In one slide!

- Application-specific topology type is derived from core FleCSI topologies through compile-time policies
- Data model of “fields” over “index spaces” is naturally data parallel
- Storage class/policy for fields (policy) can be dense or sparse
- Task-based execution model

```
flecsi_register_field(mesh_t, example, field, double, sparse, 1, mesh);

…

void ns::init(mesh<ro> mesh, sparse_field_mutator f) {

  for (auto c: mesh.cells()) {
    const size_t random = (rand()/double{RAND_MAX}) * 5;

    for (size_t i{0}; i < random; ++i) {
      const size_t entry = (rand()/double{RAND_MAX}) * 5;
      f(c, entry) = entry;
    }
  }
}
flecsi_register_task(init, ns, loc, single);

…

void driver() {
  auto m = flecsi_get_client_handle(mesh_t, clients, mesh);
  auto f = flecsi_get_mutator(m, example, field, double, sparse, 0, 5);
  flecsi_execute_task(init, ns, single, m, f);
}
```

storage class

update sparse field

get sparse mutator

FleCSI programming

In one slide!

- Application-specific topology type is derived from core FleCSI topologies through compile-time policies
- Data model of “fields” over “index spaces” is naturally data parallel
- Storage class/policy for fields (policy) can be dense or sparse
- Task-based execution model

```
flecsi_register_field(mesh_t, example, field, double, sparse, 1, mesh);

…

void ns::init(mesh<ro> mesh, sparse_field_mutator f) {

  for (auto c: mesh.cells()) {
    const size_t random = (rand()/double{RAND_MAX}) * 5;

    for (size_t i{0}; i < random; ++i) {
      const size_t entry = (rand()/double{RAND_MAX}) * 5;
      f(c, entry) = entry;
    }
  }
}
flecsi_register_task(init, ns, loc, single);

…

void driver() {
  auto m = flecsi_get_client_handle(mesh_t, clients, mesh);
  auto f = flecsi_get_mutator(m, example, field, double, sparse, 0, 5);
  flecsi_execute_task(init, ns, single, m, f);
}
```

FleCSI task

data privileges allow the runtime to infer which state changes need to be updated to maintain distributed-memory consistency

processor type

register with execution model

launch type

execution

Legion data-centric programming

Separation of concerns between parallel correctness and performance

Tasks
(execution model)

Describe parallel execution elements and algorithmic operations with sequential semantics, out-of-order execution, and in-order completion

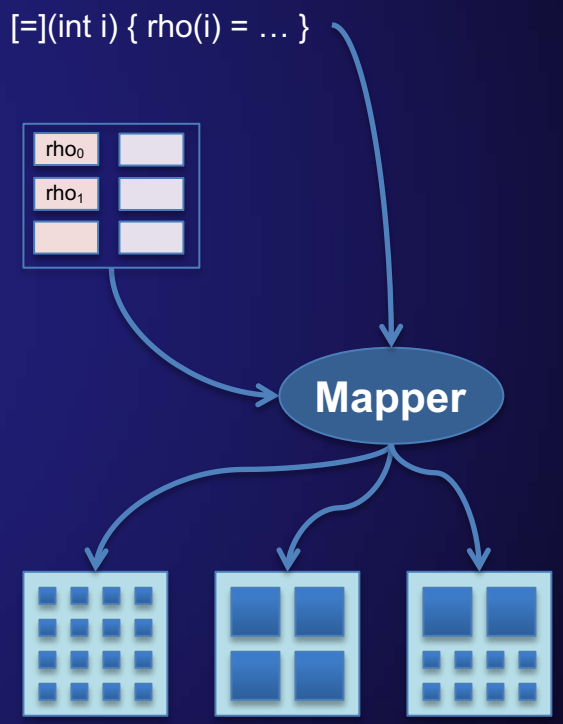
Regions
(data model)

Describe decomposition of computational domain

- **Privileges (read-write, read-only, reduce)**
- **Coherence (exclusive, atomic)**

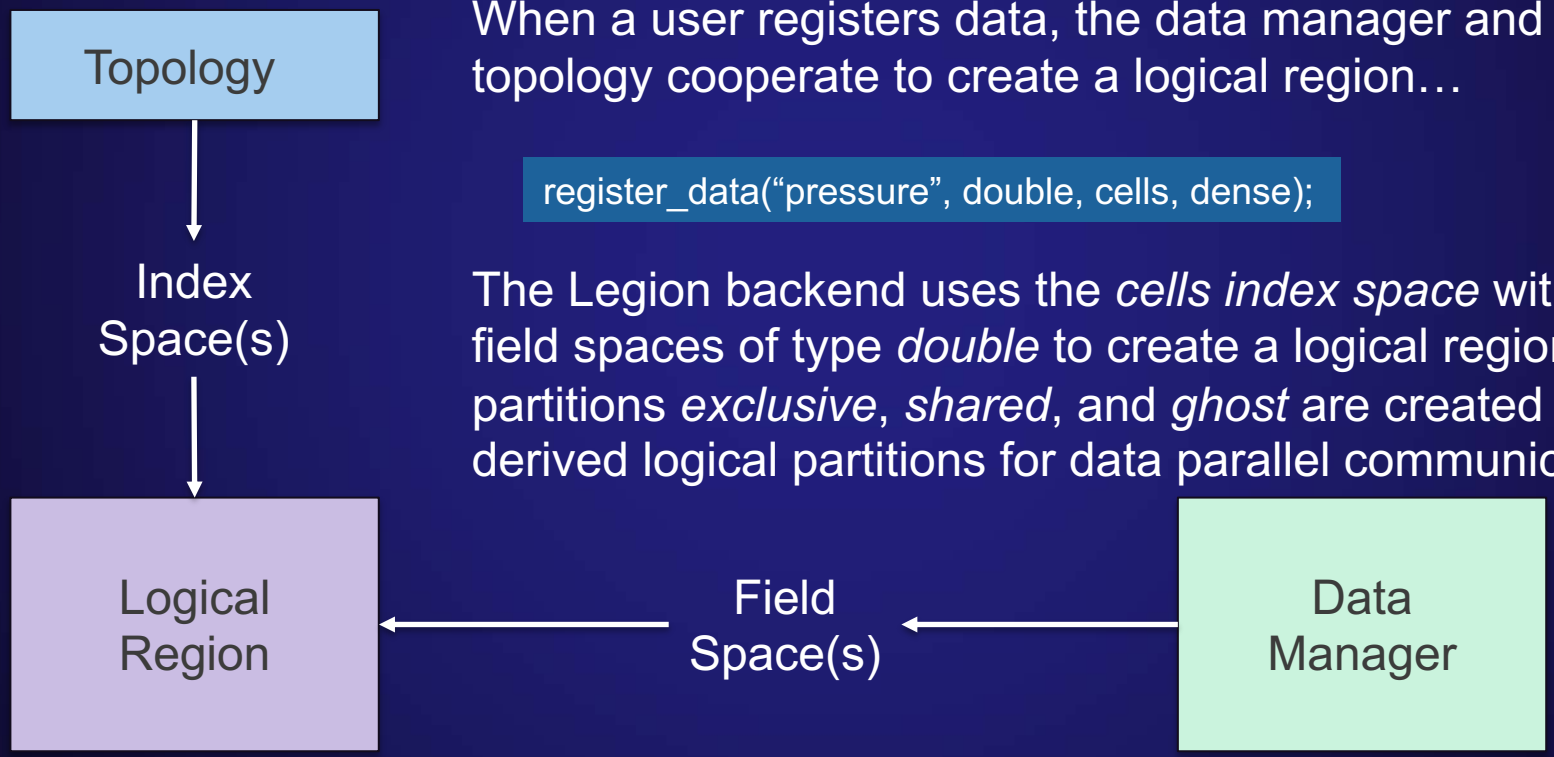
Mapper

Describes how tasks and regions should be mapped to the target architecture



Mapper allows architecture-specific optimization without effecting the correctness of the task or domain descriptions

How do the FleCSI topology and data models work together for Legion?



When a user registers data, the data manager and topology cooperate to create a logical region…

```
register_data("pressure", double, cells, dense);
```

The Legion backend uses the *cells index space* with new field spaces of type *double* to create a logical region. Index partitions *exclusive*, *shared*, and *ghost* are created as derived logical partitions for data parallel communication

FleCSI+Kokkos integration for on-node parallelism

- Challenges overcome
  - Integration into FleCSI tasks
  - Co-existence of iteration mechanisms
- OpenMP and CUDA backend
  - For CUDA, C++17 features working with Clang
- FleCSI application-specific topologies require some GPU implementation through Kokkos + CUDA
- Targeting demonstration on LLNL *Sierra* – Legion & MPI

Y = a * X

```
49 void runKokkos(){
50   Kokkos::initialize();
51   {//setting a scope for stack variables
52     const int kDataLen = 4;
53     float a = 2.0f;
54     auto x = Kokkos::DualView<float*>("x", kDataLen);
55     auto y = Kokkos::DualView<float*>("y", kDataLen);
56
57     x.h_view.data()[0] = 1.0f;
58     x.h_view.data()[1] = 2.0f;
59     x.h_view.data()[2] = 3.0f;
60     x.h_view.data()[3] = 4.0f;
61     Kokkos::parallel_for(kDataLen, KOKKOS_LAMBDA(const int i){
62       y.d_view.data()[i] = a * x.d_view.data()[i];
63     });
64     // Make sure the kernel finishes before moving on
65     Kokkos::fence();
66
67     // Print the results.
68     for (int i = 0; i < kDataLen; ++i) {
69       cout << "y[" << i << "] = " << y.h_view.data()[i] << endl;
70     }
71   }
72   Kokkos::finalize();
73   return;
```

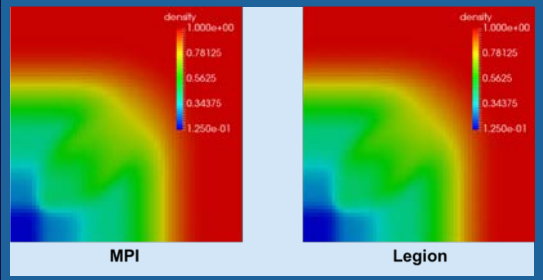
```
105 //-----------------------------------------------------------------------
106 // Loop over each vertex
107 //-----------------------------------------------------------------------
108
109 auto vertex_count = mesh.vertices(subset_t::overlapping).size();
110 real_t output[vertex_count][2];
111
112 real_t* device_output;
113 cudaMalloc(&device_output, vertex_count * sizeof(real_t) * 2);
114
115 Kokkos::parallel_for(vertex_count, KOKKOS_LAMBDA(const int temp_index){
116   const auto& vt = *(mesh.vertices(subset_t::overlapping).begin() + temp_index);
117
118   // create the final matrix the point
119   matrix_t Mp(0);
120   vector_t rhs(0);
121
122   // get the corners
123   auto cnrs = mesh.corners(vt);
124   auto num_corners = cnrs.size();
125
126   // create some corner storage
127   matrix_t[num_corners] Mpc;
128   real_t[num_corners * max_materials] weighted_impedance;
129 });
130
131 // Make sure the kernel finishes before moving on
132 Kokkos::fence();
```

Mesh vertex operations within a task in a FleCSI-based ALE fluid code

FleCSI provides portability to new technologies

Example: Legion advanced parallel runtime in place of MPI

FleCSI runtime portability



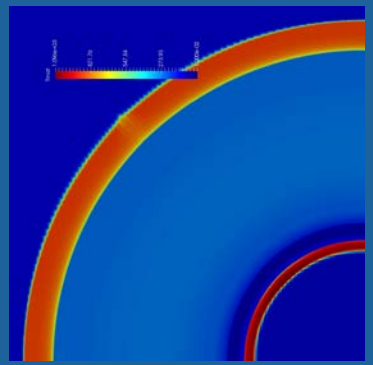
FleCSI allows users to switch between Legion and MPI backends with no change to the physics application code

FleCSI sparse storage

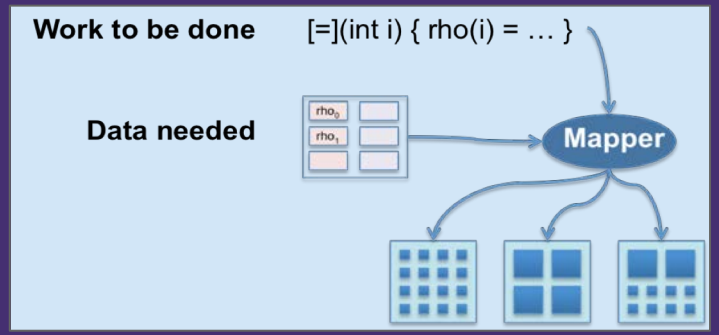
Efficient data structure for multi-materials and chemical reactions

Symphony, a Ristra multi-physics code running a simplified ICF problem.

Symphony uses FleCSI's sparse storage for efficient representation of multi-materials

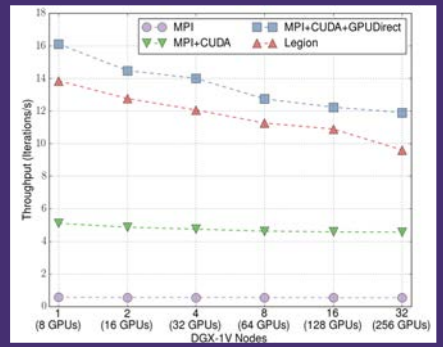


Legion data-centric programming



The *Legion* programming system separates correctness and performance. *Tasks* describe work to be done; *Regions* describe data dependencies and decomposition; *Mapper* enables architecture-specific optimization

Legion outperforms vanilla MPI+CUDA for the Pennant proxy application on up to 256 GPUs (work with NVIDIA). GPUDirect optimizations for Legion are in progress.

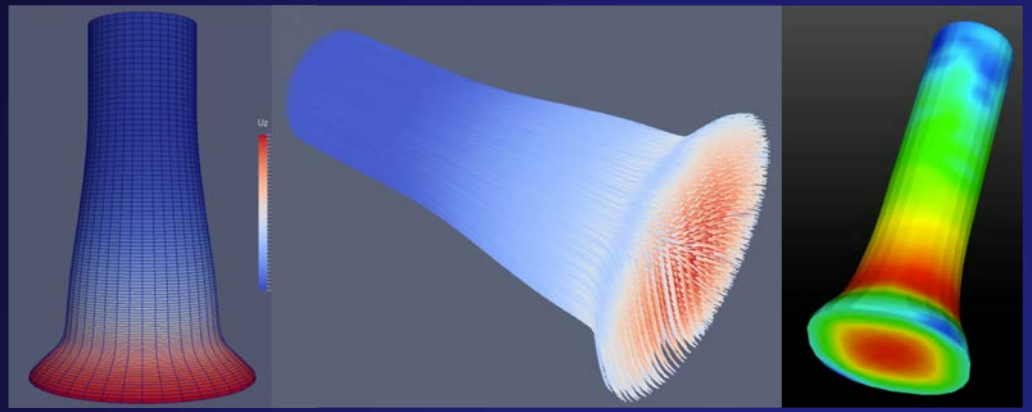


Alternative directions

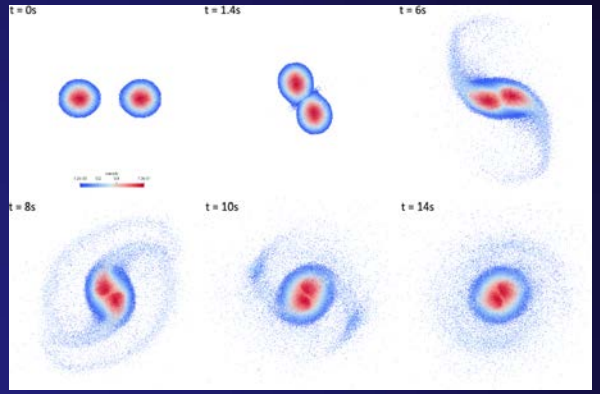
And things I didn't talk about

- Higher-order methods
- Machine learning
- Domain-specific languages
- Domain-specific architectures

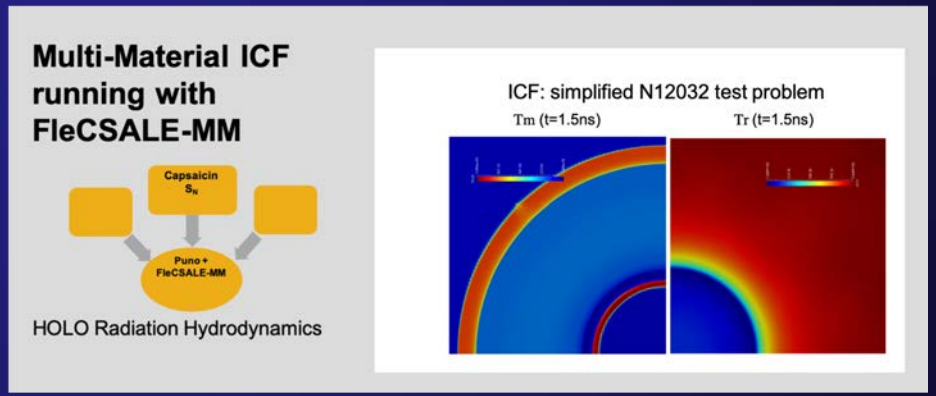
Ristra: multi-physics for an extreme-scale, heterogeneous world of computing



Taylor Anvil simulations using Hyperelastic (left), Material-Point Method (MPM) (center), and Visco-Plastic Self-Consistent (VPSC) (right) material-strength models in a FleCSI-based Lagrangian continuum dynamics code. VPSC is a micro-structure-aware strength model that represents, in a mean-field approximation, the asymmetric evolution due to grain structure (*Lebensohn, et al.*)



Binary neutron star coalescence with 40,000 particles using FleCSPH. *Loiseau et al. 2019*



Inertial confinement fusion simulations using a multi-scale radiation hydrodynamics. *Park et al. 2018.*

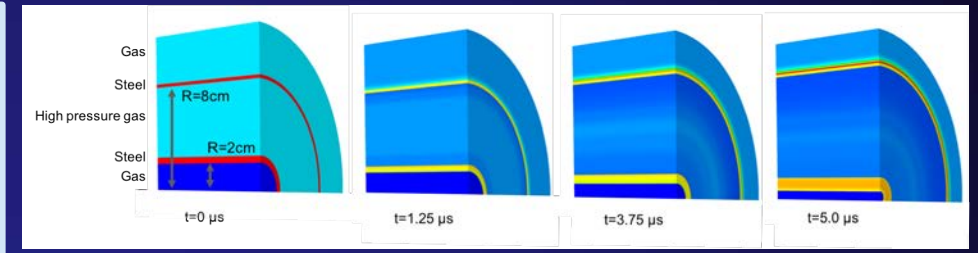
EXTRA SLIDES

The Los Alamos Ristra project (ECP and ASC)

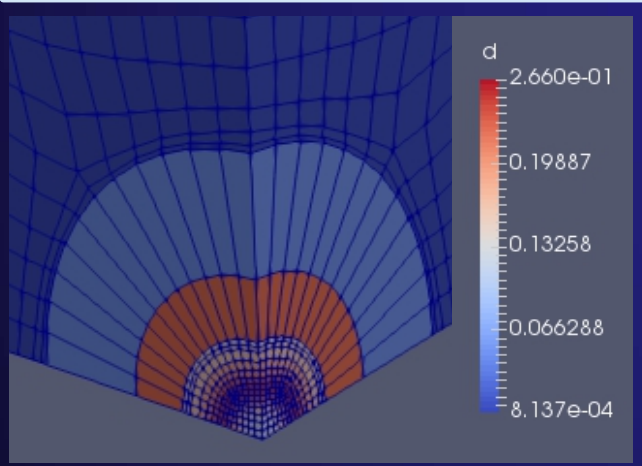
Next-generation multi-physics for basic science and national security applications

Ever changing scientific questions & Rapidly changing HPC hardware

- Ristra is creating a ***flexible and agile*** toolkit for next-generation multi-physics
- FleCSI abstraction layer separates concerns of physics and CS developers
- Portage remap and link library connects meshes, packages, and codes



Ristra *FUEL* code results for a multi-material hydrodynamic test problem with contact discontinuities, shocks, and release waves.



Ristra *Symphony* code results for an imploding ICF capsule (N12032)