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
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)

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

For physics beyond exascale, reconsider algorithmic tradeoffs
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
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

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)
• Coarse: multi-physics coupling
• Fine: higher fidelity and asynchronous concurrency

Novel programming models

Common theme at exascale: need for asynchronous 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
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 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

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

```c
...}
```

```c
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;
        }
    }
}
```

```c
flecsi_register_task(init, ns, loc, single);
```

```c
...}
```

```c
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

```c
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

```cpp
def 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);
}
```

Application-specific topology type

- 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 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

```c
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

```cpp
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

```cpp
flecsi_register_field(mesh_t, example, field, double, sparse, 1, mesh);
...
FleCSI task

void ns::init(mesh<const 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);
...
```

```cpp
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);
}
data privileges allow the runtime to infer which state changes need to be updated to maintain distributed-memory consistency

Los Alamos National Laboratory
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 affecting the correctness of the task or domain descriptions

[ bla(int i) { rho(i) = ... } ]
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

\[ Y = a \times X \]
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


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

Multi-Material ICF running with FleCSALE-MM

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

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

**Symphony**
Multi-scale, multi-material radiation hydrodynamics

**FUEL**
Multi-material hydrodynamics with material strength and realistic EOS

**FleCSALE**
Gas dynamics

**FleCSPH**
Mesh-free hydrodynamics

**Symphony**
Multi-scale, multi-material radiation hydrodynamics

**FUEL**
Multi-material hydrodynamics with material strength and realistic EOS

**FleCSALE**
Gas dynamics

**FleCSPH**
Mesh-free hydrodynamics

**Legion**
Advanced modern parallel runtime

**Legion**
Abstract models for data distribution, data execution

**MPI**
Traditional parallel runtime