FleCSI

The Flexible Computational Science Infrastructure

Developer Guide

FleCSI is developed under the Department of Energy (DOE) Advanced Technology Development and Mitigation (ATDM) Program as part of the Next Generation Architecture and Software Development (ASD) Project.

Maintainers

Ben Bergen
Applied Computer Science (CCS-7)
Los Alamos National Laboratory
bergen@lanl.gov

Marc Charest
Lagrangian Codes (XCP-1)
Los Alamos National Laboratory
charest@lanl.gov

Irina Demeshko
Applied Computer Science (CCS-7)
Los Alamos National Laboratory
irina@lanl.gov

Li-Ta (Ollie) Lo
Applied Computer Science (CCS-7)
Los Alamos National Laboratory
ollie@lanl.gov

Nick Moss
Guest Scientist
Los Alamos National Laboratory
nickdmoss@gmail.com

Navamita Ray
Applied Computer Science (CCS-7)
Los Alamos National Laboratory
nray@lanl.gov

Karen Tsai
Applied Computer Science (CCS-7)
Los Alamos National Laboratory
ktsai@lanl.gov

Wei Wu
Applied Computer Science (CCS-7)
Los Alamos National Laboratory
wwu@lanl.gov

Contributors

David Daniel
Gary Dilts
Wes Even
Rao Garimella
Jonathan Graham
Amy Hungerford
Nathaniel Morgan
Joe Schmidt
Galen Shipman
John Wohlbier

Contents

Introduction

FleCSI is a compile-time configurable C++ framework designed to support multi-physics application development. As such, FleCSI attempts to provide 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. Current support includes multi-dimensional mesh topology, mesh geometry, and mesh adjacency information, n-dimensional hashed-tree data structures, graph partitioning interfaces, and dependency closures (to identify data dependencies between distributed-memory address spaces).

FleCSI also introduces a functional programming model with control, execution, and data abstractions that are consistent state-of-the-art task-based runtimes such as Legion and Charm++. The FleCSI abstraction layer provides the developer with insulation from the underlying runtime, while allowing support for multiple runtime systems, including conventional models like asynchronous MPI. The intent is to give developers 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 architectures and runtimes that arise in the future.

FleCSI uses static polymorphism, template meta-programming techniques, and other modern C++ features to achieve high runtime performance, customizability, and to enable DSL-like features in our programming model. In both the mesh and tree topology types, FleCSI adopts a three-tiered approach: a low-level substrate that is specialized by a mid-level layer to create high-level application interfaces that hide the complexity of the underlying templated classes. This structure facilitates separation of concerns, both between developer roles, and between the structural components that make up a FleCSI-based application. As an example, for a mesh of dimension D_m, the low-level interface provides generic compile-time configurable components which deal with *entities* of varying topological dimension D_m (cell), $D_m - 1$, (face/edge), etc. Each of these entities resides in a *domain* M or sub-mesh. Entities are connected to each other by a *connectivity* using a compressed id/offset representation for efficient space utilization and fast traversal.

Code Structure

Name Spaces

FleCSI uses several different namespaces:

- **data**
  Data model types.
- **execution**
  Execution model types.
- **topology**
  Topology types.
- **io**
  I/O types.

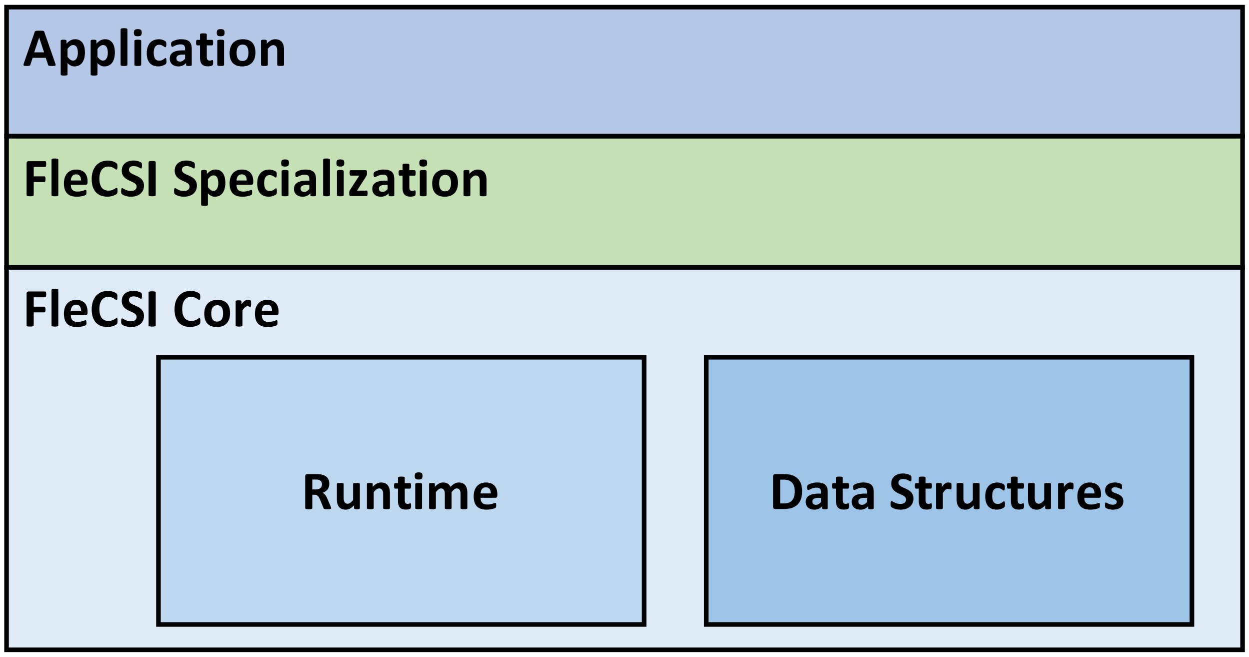

Figure 1:

- **utils**
  Utilities.

Index Spaces

The FleCSI data and execution models are premised on the notion of index spaces. Simply put, an index space is an enumerated set. An index space can be defined in several ways. For example, an index space with which many people are familiar is a range. Consider the loop

```
for(int i=0; i<5; ++i) { std::cout << "i=" << i << std::endl; }
```

Implicitly, the values that are taken on by i form an index space, i.e., the set of enumerated indices

```
{ 0, 1, 2, 3, 4 }
```

A more relevant example is the set of indices for the canonical triangle mesh in Figure 2. Both the cells and the vertices represent index spaces. The cells index space has 29 objects, indexed from 1 to 29

```
{ 1,  2,  3,  4,  5,  6,  7,  8,  9,  10, 11, 12, 13, 14,
  15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29 },
```

while the vertex index space has 23 objects, indexed from 1 to 23

```
{ 1,  2,  3,  4,  5,  6,  7,  8,  9,  10, 11, 12, 13, 14,
  15, 16, 17, 18, 19, 20, 21, 22, 23 }.
```

A basic understanding of index spaces and these examples is critical to understanding the FleCSI data model.

Formally, a space is a set of indices with some added structure. An enumerated set satisfies this definition. However, the enumerated sets in FleCSI are better classified as topological

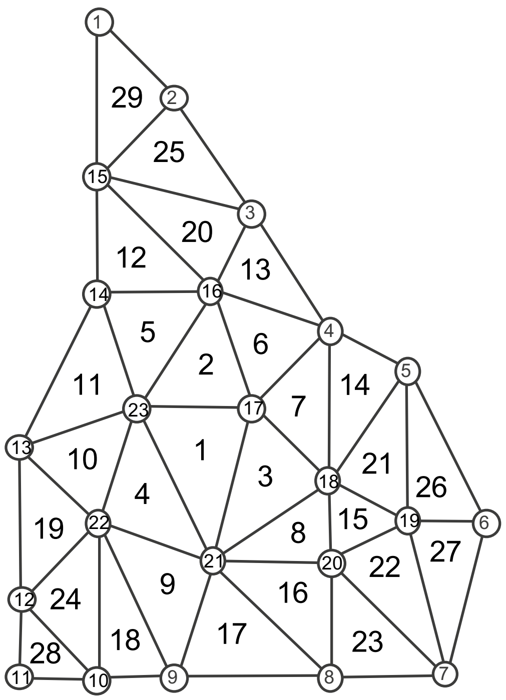

Figure 2: A canonical triangle mesh example with cells and vertices.

spaces. In general, FleCSI index spaces–discussed here and below–represent discrete topologies. Colloquilly, index spaces may also be referred to as index sets.

Subsets

Given an index space, we can form subsets that only contain some of the objects of the original set, e.g.,

```
{ 4, 9, 10, 11, 16, 17, 18, 19, 24, 28 }
```

is a proper subset of the cells index space in Figure 2

```
{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
  17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29 }
```

Subsets are a useful way to define distributed-memory colorsings. Consider the disjoint partitioning of our canonical mesh in Figure 3. We can represent this coloring by defining the index spaces

```
1. { 4, 9, 10, 11, 16, 17, 18, 19, 24, 28 }
2. { 1, 2, 5, 6, 12, 13, 20, 25, 29 }
3. { 3, 7, 8, 14, 15, 21, 22, 23, 26, 27 }
```

Data Model

The FleCSI data model provides an intuitive high-level user interface that can be specialized using different low-level storage types. Each storage type provides data registration, data handles, and data mutators (when

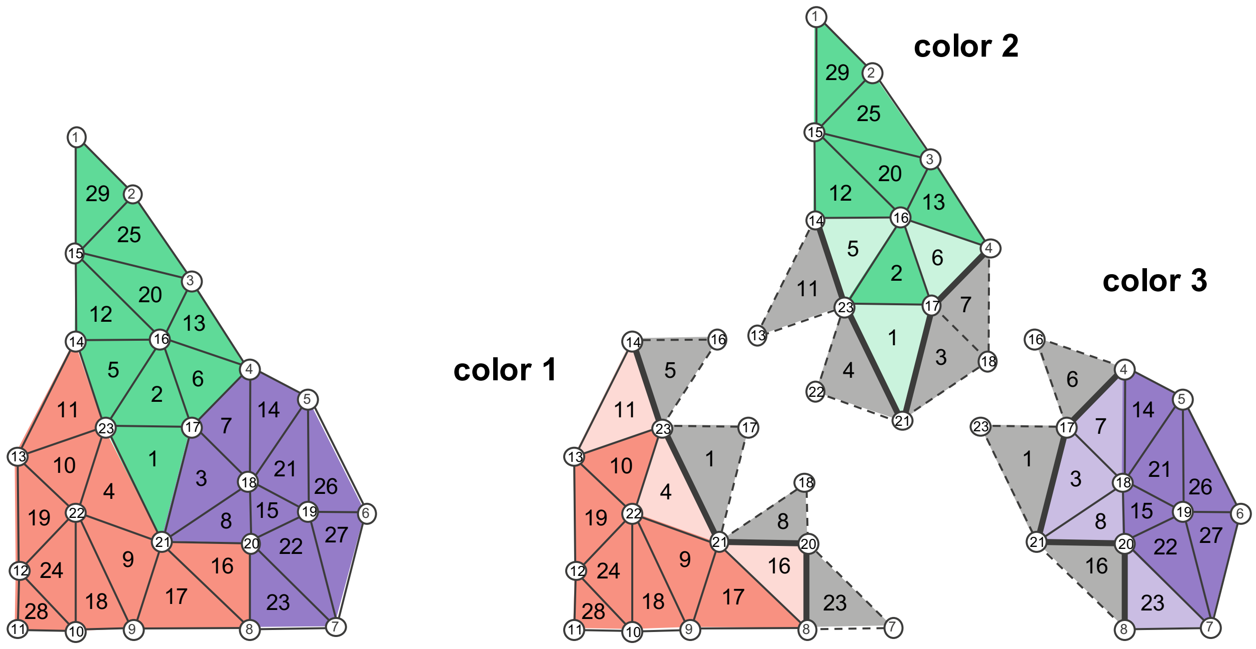

Figure 3: Disjoint coloring of our canonical triangle mesh.

appropriate) that allow the user to modify data values and structure. The currently supported storage types are:

- **dense**
  This storage type provides a one-dimensional dense array suitable for storing structured or unstructured data.
- **sparse**
  This storage type provides compressed storage for a logically dense index space.
- **global**
  This storage type is suitable for storing data that are non-enumerable, i.e., data that are not logically stored as an array.
- **color**
  This storage type is suitable for storing non-enumerable data on a *per-color* basis.
- **tuple**
  This storage type provides c-struct-like support so that task arguments can be associated without forcing a particular data layout.

Execution Model Overview

FleCSI's execution model is hierarchical and data-centric, with work being done through tasks and kernels:

- **task**
  Tasks operate on logically distributed address spaces with output determined only by inputs and with no observable side effects, i.e., they are pure functions.
- **kernel**
  Kernels operate on logically shared memory and may have side effects within their local address space.

Both of these work types may be executed in parallel, so the attributes that distinguish them from each other are best considered in the context of memory consitency: Tasks work on locally-mapped data that are logically distributed. Consistency of the distributed data state is maintained by the FleCSI runtime. Each task instance always executes within a single address space. Kernels work on logically shared data with a relaxed consistency model, i.e., the user is responsible for implementing memory consistency by applying synchronization techniques. Kernels may only be invoked from within a task.

A simple, but incomplete view of this model is that tasks are internode, and kernels are intranode, or, more precisely, intraprocessor. A more complete view is much less restrictive of tasks, including only the constraint that a task be a pure function.

Consider the following:

```
double update(mesh<ro> m, field<rw, rw, ro> p) {
  double sum{0};

  forall(auto c: m.cells(owned)) {
    p(c) = 2.0*p(c);
    sum += p(c);
  } // forall

  return sum;
} // task
```

In this example, the *update* function is a task and the *forall* loop implicitly defines a kernel. Ignoring many of the details, we see that the task is *mostly* semantically sequential from a distributed-memory point of view, i.e., it takes a mesh *m* and a field *p*, both of which may be distributed, and applies updates as if the entire index space were available locally. The admonishment that the task is *mostly* semantically sequential refers to the *owned* identifier in the loop construct. Here, *owned* indicates that the mesh should return an iterator only to the exclusive and shared cell indices (see the discussion on index spaces). Implicitly, this exposes the distributed-memory nature of the task.

The *forall* kernel, on the other hand, is explicitly data parallel. The iterator *m.cells(owned)* defines an index space over which the loop body may be executed in parallel. This is a very concise syntax that is supported by the FleCSI C++ language extensions (The use of the term *kernel* in our nomenclature derives from the CUDA and OpenCL programming models, which may be familiar to the reader.)

Execution Model

The FleCSI execution model is implemented through a combination of macro and C++ interfaces. Macros are used to implement the high-level user interface through calls to the core C++ interface. The general structure of the code is illustrated in figure FIXME.

The primary interface classes context_t, task_model_t, and function_t are types that are defined during configuration that encode the low-level runtime that was selected for the build. These types are then used in the macro definitions to implement the high-level FleCSI interface. Currently, FleCSI supports the MPI and Legion distributed-memory runtimes. The code to implement the backend implementations for each of these is in the respectively named sub-directory of *execution*, e.g., the Legion implementation is in *legion*. Documentation for the macro and core C++ interfaces is maintained in the Doxygen documentation.

Note: Compile-time selection of the low-level runtime is handled by the pre-processor through type definition in files of the form *flecsi_runtime_X*, where *X* is a policy or runtime model, e.g., flecsi_runtime_context_policy, or flecsi_runtime_execution_policy. These should be updated when new backend support is added or when a runtime is removed. The files are located in the *flecsi* sub-directory.

Tasks

FleCSI tasks are pure functions with controlled side-effects. This variation of *pure* is required to allow runtime calls from within an executing task. In general, data states are never altered by the runtime, although they may be moved or managed. Any such changes executed by the runtime will be transparent to the task and will not alter correctness.

Functions

The FleCSI function interface provides a mechanism for creating relocatable function references in the form of a function handle. Function handles are first-class objects that may be passed as data. They are functionally equivalent to a C++ std::function.

Kernels

FleCSI kernels are implicitly defined by data-parallel semantics. In particular, the FleCSI C++ language extensions add the *forall* keyword. Logically, each occurance of a forall loop defines a kernel that can be applied to a given index space. This construct has relaxed memory consistency and is similar to OpenCL or CUDA kernels, or to pragmatized OpenMP loops.

FIXME: Working notes for structured mesh interface

Neighbor information can be specified with from_dim, to_dim, and thru_dim where:

- from_dim: The topological dimension of the entity for which neighbors should be found.
- to_dim: The topological dimension of the neighbors.
- thru_dim: The intermediate dimension across which the neighbor must be found, e.g.:

| 6 | 7 | 8 |

| 3 | 4 | 5 |

| 0 | 1 | 2 | ———-

The neighbors of cell 0 thru dimension 0 are: 1, 3, 4 The neighbors of cell 0 thru dimension 1 are: 1, 3

The interface for the structured mesh should provide a connectivities interface that uses this information, e.g.:

```
template<size_t from_dim, size_t to_dim, size_t thru_dim>
iterator_type (to be defined)
connectivities(
  size_t id
)
{
} // connectivities
```

Topology Types

FleCSI provides data structures and algorithms for several mesh and tree types that are all based on a common design pattern, which allows static specialization of the underlying data structure with user-provided *entity* type definitions. The user specified types are defined by a *policy* that specializes the low-level FleCSI data structure. The following sections describe the currently supported types.

Mesh Topology

The low-level mesh interface is parameterized by a policy, which defines various properties such as mesh dimension, and concrete entity classes corresponding to each domain and topological dimension. The mesh policy defines a series of tuples in order to declare its entity types for each topological dimension and domain, and select connectivities between each entity. FleCSI supports a specialized type of localized connectivity called a *binding*, which connects entities from one domain to another domain.

FleCSI separates mesh topology from geometry, and the mesh–from the topology's perspective–is simply a connected graph. Vertex coordinates and other application data are part of the *state model*. Our connectivity computation algorithms are based on DOLFIN. Once vertices and cells have been created, the remainder of the connectivity data is computed automatically by the mesh topology through the following three algorithms: *build*, *transpose*, and *intersect*, e.g., *build* is used to compute edges using cell-to-vertex connectivity and is also responsible for creating entity objects associated with these edges. From a connectivity involving topological dimensions $D_1 \to D_2$, transpose creates connectivity $D_2 \to D_1$. Intersect, given $D_1 \to D'$ and $D' \to D_2$, computes $D_1 \to D_2$.

The low-level mesh topology provides a set of iterable objects that a mid-level specialization can make available to an application to allow, at a high-level, iteration through connectivities using an intuitive *ranged-based for* syntax, e.g., forall cells c_i, forall edges e_i of cell c_i. Entities can be stored in sets that also support range-based for iterations and enable set operations such as union, intersection, difference, and provide functional model capabilities with *filter*, *apply*, *map*, *reduce*, etc.

N-Tree Topology

The tree topology, applying the philosophy of mesh topology, supports a D-dimensional tree topology, e.g., for $D = 3$, an octree. Similar to meshes, the tree topology is parameterized on a policy that defines its branch and entity types where branches define a container for entities at the leafs as well as refinement and coarsening control. A tree geometry class is used that is specialized for dimension and handles such things as distance or intersection in locality queries. Branch id's are mapped from geometrical coordinates to an integer using a Morton id scheme so that branches can be stored in a hashed/unordered map–and, additionally, pointers to child branches are stored for efficient recursive traversal. Branch hashing allows for efficient random access, e.g., given arbitrary coordinates, find the parent branch at the current deepest level of the tree, i.e., the child's insertion branch. Refinement and coarsening are delegated to the policy, and when an application requests a refinement, the policy has control over when and whether the parent branch is finally refined. Entity and branch sets support the same set operations and functional operations as mesh topology. C++11-style callable objects are used to make methods on the tree generic but efficient. Various methods allow branches and entities to be visited recursively. When an entity changes position, we allow an entity's position in the tree to be updated as efficiently as possible without necessarily requiring a reinsertion.

K-D Tree Topology

I/O

Unit Tests

Unit tests should only be used to test specific components of FleCSI, e.g., types or algorithms. **They shall not be used to develop new applications!**. Sometimes, it *is* necessary to write a unit test that depends on the FleCSI library itself. Developers should try to avoid this dependence! However, when it is necessary, it is likely that such a unit test will have unresolved symbols from the various runtime libraries to which the FleCSI library links. To resolve these, the developer should add ${CINCH_RUNTIME_LIBRARIES} to the LIBRARIES argument to the cinch_add_unit() function:

```
cinch_add_unit(mytest,
  SOURCES
    test/mytest.cc
  LIBRARIES
    flecsi ${CINCH_RUNTIME_LIBRARIES}
)
```

In addition to ${CINCH_RUNTIME_LIBRARIES}, ${CINCH_RUNTIME_INCLUDES}, and ${CINCH_RUNTIME_FLAGS} are also included.

Runtime Initialization Structure

Internally, the FleCSI runtime goes through several initialization steps that allow both the specialization and FleCSI types that are part of the runtime to execute control point logic. The current structure of this initialization is shown in Figure 4.

C++ Language Extensions

FleCSI provides fine-grained, data-parallel semantics through the introduction of the several keywords: *forall*, *reduceall*, and *scan*. These are extensions to standard C++ syntax. As an example, consider the following task definition:

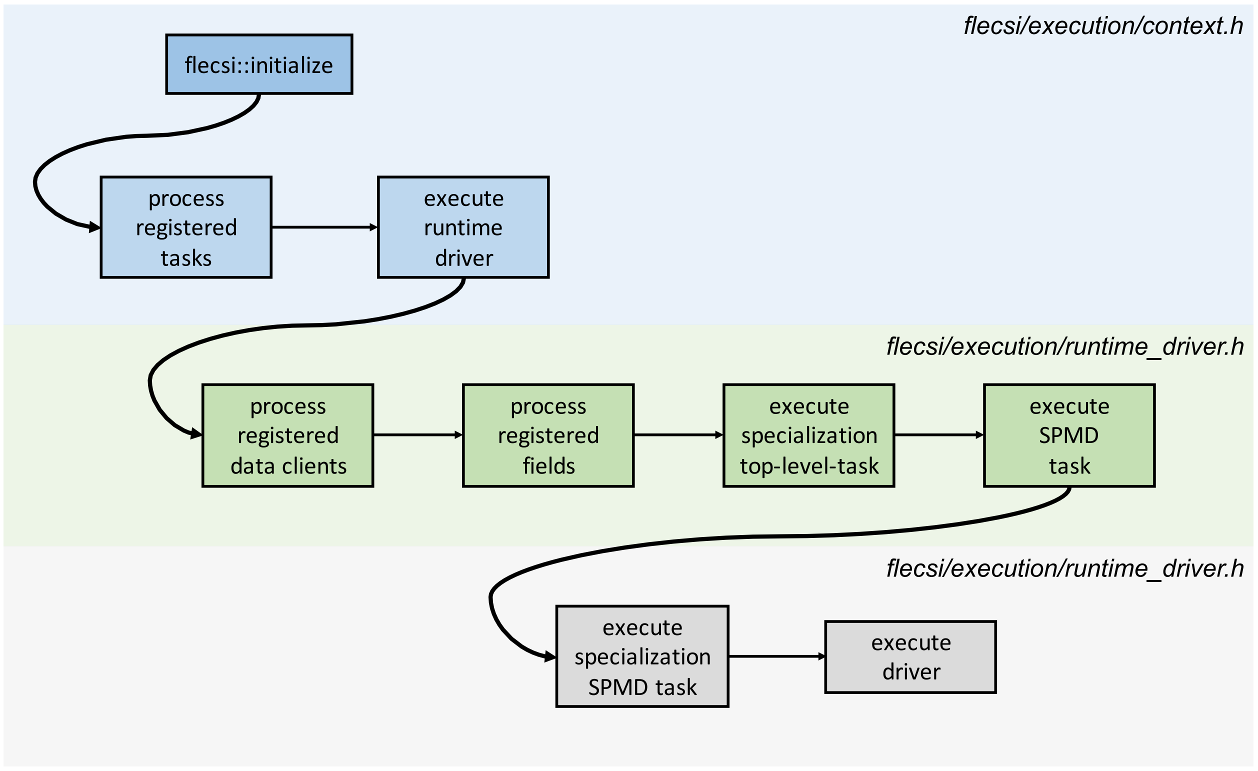
flecsi/execution/context.h
flecsi::initialize
process registered tasks
execute runtime driver
flecsi/execution/runtime_driver.h
process registered data clients
process registered fields
execute specialization top-level-task
execute SPMD task
flecsi/execution/runtime_driver.h
execute specialization SPMD task
execute driver

Figure 4: FleCSI Runtime Initialization Structure.

```
void update(mesh<ro> m, field<rw, rw, ro> p) {
  forall(auto c: m.cells(owned)) {
    p(c) += 1.0;
  } // forall
} // update
```

The forall loop defines a FleCSI *kernel* that can be executed in parallel.

The descision to modify C++ is partially motivated by a desire to capture parallel information in a direct way that will allow portable optimizations. For example, on CPU architectures, the introduction of loop-carried dependencies is often an important optimization step. A pure C++ implementation of forall would inhibit this optimization because the iterates of the loop would have to be executed as function object invocations to comply with C++ syntax rules. The use of a language extension allows our compiler frontend to make optimization decisions based on the target architecture, and gives greater latitude on the code transformations that can be applied.

Another motivation is the added ability of our approach to integrate loop-level parallelism with knowledge about task execution. Task registration and execution in the FleCSI model explicitly specify on which processor type the task shall be executed. This information can be used in conjuntion with our parallel syntax to identify the target architecture for which to optimize. Additionally, because tasks are pure functions, any data motion required to reconcile depenencies between separate address spaces can be handled during the task prologue and epilogue stages. The Legion runtime also uses this information to hide latency. This approach provides a much better option for making choices about execution granulatiry that may affect data dependencies. As an example of why this is useful, consider the contrasting example of a direct C++ implementation of a forall interface:

In this case, we are effectively limited to expressing parallelism and data movement at the level of the forall loop. In particular, if data must be offloaded to an accelerator device to perform the loop body, the overhead of that operation must be compensated for by the arithmetic complexity of the loop logic for that single loop. Additionally, a policy must also be specified as part of the signature of the interface. This adds noise to the code that is avoidable.

Using a combination of FleCSI tasks and kernels allows the user to separate the concerns of fine-grained data parallelism and distributed-memory data dependencies:

```
// Task
//
// Mesh and field data will be resident in the correct
// address space by the time the task is invoked.
void update(mesh<ro> m, field<rw, rw, ro> p) {

  // Each of the following kernels can execute in parallel
  // if parallel execution is supported by the target architecture.

  // Kernal 1
  forall(auto c: m.cells(owned)) {
    p(c) += 1.0;
  } // forall

  // Kernal 2
  forall(auto c: m.cells(owned)) {
    p(c) *= 2.0;
  } // forall

  // Kernal 3
  forall(auto c: m.cells(owned)) {
    p(c) -= 1.0;
  } // forall

} // update
```

Tasks have low overhead, so if the user really desires loop-level parallelization, they can still acheive this by creating a task with a single forall loop. However, the distinction between tasks and kernels provides a good mechanism for reasoning about data dependencies between parallel operations.

Mesh Coloring

FleCSI supports the partitioning or *coloring* of meshes through a set of core interface methods that can be accessed by a specialization. Using these methods, the specialization can generate the independent entity coloring, and colorings for dependent entities, e.g., a mesh that has a primary or *independent* coloring based on the cell adjacency graph, with the *dependent* coloring of the vertices generated from the cell coloring by applying a rule such as lowest color ownership to assign the vertex owners.

Vim Syntax Highlighting

The FleCSI C++ language extensions add new syntax that can be highlighted in vim. To install the files, simply execute the *install* script in this directory:

```
$ ./install
```

The files will be copied into the appropriate location in your home directory. When you open a file with the FleCSI *.fcc* or *.fh* file, your code will be correctly highlighted.