The Flexible Computational Science Infrastructure

Developer Guide

FleCSI is developed under the Department of Energy (DOE) Advanced Technology Development and Mitigration (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
Applied Computer Science (CCS-7)
Los Alamos National Laboratory
nickm@lanl.gov

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

Building FleCSI

FleCSI can be configured to run with different distributed-memory runtimes, including Legion, and MPI. FleCSI also has support for various fine-grained, node-level runtimes, including OpenMP, Kokkos, Agency, and the C++17 extensions for parallelism. Full documentation of FleCSI requires both Pandoc and Doxygen. These configuration options are listed to convey to the reader that the FleCSI build system has several paths that can be taken to tailor FleCSI to a given system and architecture.

Requirements & Prerequisites

The following list of requirements provides a complete set of build options, but is not necessary for a particular build:

- **C++14 compliant compiler**
  At the current time, FleCSI has been tested with GNU, Clang, and Intel C++ compilers.

- **MPI**
  If Legion support is enabled, the MPI implementation must have support for *MPI_THREAD_MULTIPLE*.
- **Legion**
  We are currently using the most up-to-date version of the master branch.
- **GASNet**
  GASNet is only required if Legion support is enabled.
- **Pandoc**
  Pandoc is only required to build the FleCSI guide documentation. Pandoc is a format conversion tool. More information is available at http://pandoc.org.
- **Doxygen**
  Doxygen is only required to build the interface documentation.
- **CMake**
  We currently require CMake version 2.8 or greater.
- **Python**
  We currently require Python 2.7 or greater.

FleCSI Third Party Libraries Project

To facilitate FleCSI adoption by a broad set of users, we have provided a superbuild project that can build many of the libraries and tools required to build and use FleCSI. The FleCSI Third Party Libraries (TPL) project is available from github at https://github.com/laristra/flecsi-third-party. Note that a suitable version of MPI is required for the superbuild.

Admonishment: Users should note that, while this approach is easier, it may not provide as robust a solution as individually building each dependency, and that care should be taken before installing these libraries on a production system to avoid possible conflicts or duplication. Production deployments of some of these tools may have architecture or system specific needs that will not be met by our superbuild. Users who are working with development branches of FleCSI are encouraged to build each package separately.

Build instructions for the TPLs:

```
$ git clone --recursive https://github.com/laristra/flecsi-third-party.git
$ cd flecsi-third-party
$ mkdir build
$ cd build
$ cmake -DCMAKE_INSTALL_PREFIX=/path/to/install/directory ..
$ make
```

The *make* command will also install the TPLs in the specified install directory. **It is recommended that users remove the install directory before updating or re-compiling the TPLs.**

Build Environment

FleCSI uses CMake as part of its build system. A convenient mechanism for identifying directory paths that should be searched by CMake's *find_package* function is to set the *CMAKE_PREFIX_PATH* environment variable. If you are using the FleCSI TPLs discussed above, you can set *CMAKE_PREFIX_PATH* to include the path to your TPL installation directory and FleCSI will automatically find all of its dependencies:

```
$ export CMAKE_PREFIX_PATH=/path/to/install/directory (bash)
```

Getting The Code

Clone the FleCSI git repository, and create an out-of-source build area (FleCSI prohibits in-source builds):

```
$ git clone --recursive https://github.com/laristra/flecsi.git
$ cd flecsi
$ mkdir build
$ cd build
```

Note: FleCSI developers, i.e., those who have permission to create new branches and pull requests, should clone the source code using their github account and the *git* user:

```
$ git clone --recursive git@github.com:laristra/flecsi.git
$ cd flecsi
$ mkdir build
$ cd build
```

Configuration & Build

Configuration of FleCSI requires the selection of the backend runtimes that will be used by the FleCSI programming model abstraction to invoke tasks and kernels. There are currently two supported distributed-memory runtimes, a serial runtime, and one supported node-level runtime:

- **Distributed-Memory**
  Legion or MPI
- **Serial [supported thorugh MPI runtime]**
  The serial build is no longer supported. Users wishing to emulate this build mode should select the MPI runtime and run executables with a single-rank.
- **Node-Level**
  OpenMP

Example configuration: **MPI**

```
$ cmake -DFLECSI_RUNTIME_MODEL=mpi -DENABLE_MPI -DENABLE_COLORING ..
```

Example configuration: **MPI + OpenMP**

```
$ cmake -DFLECSI_RUNTIME_MODEL=mpi -DENABLE_MPI -DENABLE_COLORING -DENABLE_OPENMP ..
```

Example configuration: **Legion**

```
$ cmake -DFLECSI_RUNTIME_MODEL=legion -DENABLE_MPI -DENABLE_COLORING ..
```

After configuration is complete, just use *make* to build:

```
$ make -j 16
```

This will build all targets *except* for the Doxygen documentation, which can be built with:

```
$ make doxygen
```

Installation uses the normal *make install*, and will install FleCSI in the directory specified by CMAKE_INSTALL_PREFIX:

```
$ make install
```

CMake Configuration Options

The following set of options are available to control how FleCSI is built.

- **BUILD_SHARED_LIBS [default: ON]**
  Build shared library objects (as opposed to static).
- **CMAKE_BUILD_TYPE [default: Debug]**
  Specify the build type (configuration) statically for this build tree. Possible choices are *Debug*, *Release*, *RelWithDebInfo*, and *MinSizeRel*.
- **CMAKE_INSTALL_PREFIX [default: /usr/local]**
  Specify the installation path to use when *make install* is invoked.
- **CXX_CONFORMANCE_STANDARD [default: c++14]**
  Specify to which C++ standard a compiler must conform. This is a developer option used to identify whether or not the selected C++ compiler will be able to compile FleCSI, and which (if any) tests it fails to compile. This information can be shared with vendors to identify features that are required by FleCSI that are not standards-compliant in the vendor's compiler.
- **ENABLE_BOOST_PREPROCESSOR [default: ON]**
  Boost.Preprocessor is a header-only Boost library that provides enhanced pre-processor options and manipulation, which are not supported by the standard C preprocessor. Currently, FleCSI uses the preprocessor to implement type reflection.
- **ENABLE_BOOST_PROGRAM_OPTIONS [default: OFF]**
  Boost.Program_options provides support for handling command-line options to a program. When this build option is enabled, CMake will attempt to locate a valid installation of the program options library, and Cinch will enable certain command-line options for unit tests. In particular, if Cinch's clog extensions are enabled, the *–tags* command-line option will be available to select output tags.
- **ENABLE_CINCH_DEVELOPMENT [default: OFF]**
  If this option is enabled, extra information will be generated to help debug different Cinch behaviors. Currenlty, this only affects the generation of documentation: when enabled, the resulting PDF documentation will be annotated with the original locations of the content. **FIXME: We should consider renaming this option**
- **ENABLE_CINCH_VERBOSE [default: OFF]**
  If this option is enabled, extra information will be output during the CMake configuration and build that may be helpful in debugging Cinch.
- **ENABLE_CLOG [default: OFF]**
  Enable Cinch Logging (clog). The Cinch logging interface provides methods for generating and controlling output from a running application.
- **CLOG_COLOR_OUTPUT [default: ON]**
  Enable colorization of clog output.
- **CLOG_DEBUG [default: OFF]**
  Enable verbose debugging output for clog.
- **CLOG_ENABLE_EXTERNAL [default: OFF]**
  The Cinch clog facility is a runtime. As such, some of the features provided by clog require initialization. Because of the C++ mechanism used by clog to implement parts of its interface, it is possible to call the interface from parts of the code that are *external*, i.e., at file scope. Externally scoped statements are executed before the clog runtime can be initialized, and therefore their output cannot be controlled by the clog tagging feature. This option allows the user to enable this type of output, which can be quite verbose.

- **CLOG_ENABLE_TAGS [default: OFF]**
  Enable the tag feature for clog. If enabled, users can selectively control clog output by specifying active tags on the command line:

  ```
  $ ./executable --tags=tag1,tag2
  ```

  Invoking the *–tags* flag with no arguments will list the available tags. **This option requries that ENABLE_BOOST_PROGRAM_OPTIONS be ON.**

- **CLOG_STRIP_LEVEL [default: 0]**
  Strip levels are another mechanism to allow the user to control the amount of output that is generated by clog. In general, the higher the stip level, the fewer the number of clog messages that will be output. There are five strip levels in clog: *trace*, *info*, *warn*, *error*, and *fatal*. Output for all of these levels is turned on if the strip level is 0. As the strip level is increased, fewer levels are output, e.g., if the strip level is 3, only *error* and *fatal* log messages will be output. **Regardless of the strip level, clog messages that are designated *fatal* will generate a runtime error and will invoke std::exit.**

- **ENABLE_COLORING [default: OFF]**
  This option controls whether or not various library dependencies and code sections are active that are required for graph partitioning (coloring) and distributed-memory parallelism. In general, if you have selected a runtime mode that requires this option, it will automatically be enabled.

- **ENABLE_COLOR_UNIT_TESTS [default: OFF]**
  Enable coloraization of unit test output.

- **ENABLE_COVERAGE_BUILD [default: OFF]**
  Enable build mode to determine the code coverage of the current set of unit tests. This is useful for continuous integration (CI) test analysis.

- **ENABLE_DEVEL_TARGETS [default: OFF]**
  Development targets allow developers to add small programs to the FleCSI source code for testing code while it is being developed. These programs are not intended to be used as unit tests, and may be added or removed as the code evolves.

- **ENABLE_DOCUMENTATION [default: OFF]**
  This option controls whether or not the FleCSI user and developer guide documentation is built. If enabled, CMake will generate these guides as PDFs in the *doc* subdirectory of the build.

- **ENABLE_DOXYGEN [default: OFF]**
  If enabled, CMake will verify that a suitable *doxygen* binary is available on the system, and will add a target for generating Doxygen-style interface documentation from the FleCSI source code. **To build the doxygen documentation, users must explicitly invoke:**

  ```
  $ make doxygen
  ```

- **ENABLE_DOXYGEN_WARN [default: OFF]**
  Normal Doxygen output produces many pages worth of warnings. These are distracting and overly verbose. As such, they are disabled by default. This options allows the user to turn them back on.

- **ENABLE_EXODUS [default: OFF]**
  If enabled, CMake will verify that a suitable Exodus library is available on the system, and will enable Exodus functionality in the FleCSI I/O interface.

- **ENABLE_FLECSIT [default: OFF]**
  FleCSIT is a command-line utility that simplifies experimental development using FleCSI. This can be thought of as a *sandbox* mode, where the user can write code that utilizes a particular FleCSI specialization and the FleCSI data and runtime models without the overhead of a full production code project. This option simply enables creation of the FleCSIT executable.

- **ENABLE_JENKINS_OUTPUT [default: OFF]**
  If this options is on, extra meta data will be output during unit test invocation that may be used by the Jenkins CI system.
- **ENABLE_MPI_CXX_BINDINGS [default: OFF]**
  This option is a fall-back for codes that actually require the MPI C++ bindings. **This interface is deprecated and should only be used if it is impossible to get rid of the dependency.**
- **ENABLE_OPENMP [default: OFF]**
  Enable OpenMP support. If enabled, the appropriate flags will be passed to the C++ compiler to enable language support for OpenMP pragmas.
- **ENABLE_OPENSSL [default: OFF]**
  If enabled, CMake will verify that a suitable OpenSSL library is available on the system, and will enable the FleCSI checksum interface.
- **ENABLE_UNIT_TESTS [default: OFF]**
  Enable FleCSI unit tests. If enabled, the unit test suite can be run by invoking:

  ```
  $ make test
  ```
- **FLECSI_COUNTER_TYPE [default: int32_t]**
  Specify the C++ type to use for the FleCSI counter interface.
- **FLECSI_DBC_ACTION [default: throw]**
  Select the design-by-contract action.
- **FLECSI_DBC_REQUIRE [default: ON]**
  Enable DBC pre/post condition assertions.
- **FLECSI_ID_FBITS [default: 4]**
  Specify the number of bits to be used to represent id flags. This option affects the number of entities that can be represented on a FleCSI mesh type. The number of bits used to represent entities is 62-FLECSI_ID_PBITS-FLECSI_ID_FBITS. With the current defaults there are 38 bits available to represent entities, i.e., up to 274877906944 entities can be resolved.
- **FLECSI_ID_PBITS [default: 20]**
  Specify the number of bits to be used to represent partition ids. This option affects the number of entities that can be represented on a FleCSI mesh type. The number of bits used to represent entities is 62-FLECSI_ID_PBITS-FLECSI_ID_FBITS. With the current defaults there are 38 bits available to represent entities, i.e., up to 274877906944 entities can be resolved.
- **FLECSI_RUNTIME_MODEL [default: mpi]**
  Specify the low-level runtime model. Currently, *legion* and *mpi* are the only valid options.
- **VERSION_CREATION [default: git describe]**
  This options allows the user to either directly specify a version by entering it here, or to let the build system provide a version using git describe.

Code Structure

Name Spaces

FleCSI uses several different namespaces:

- **data**
  Data model types.

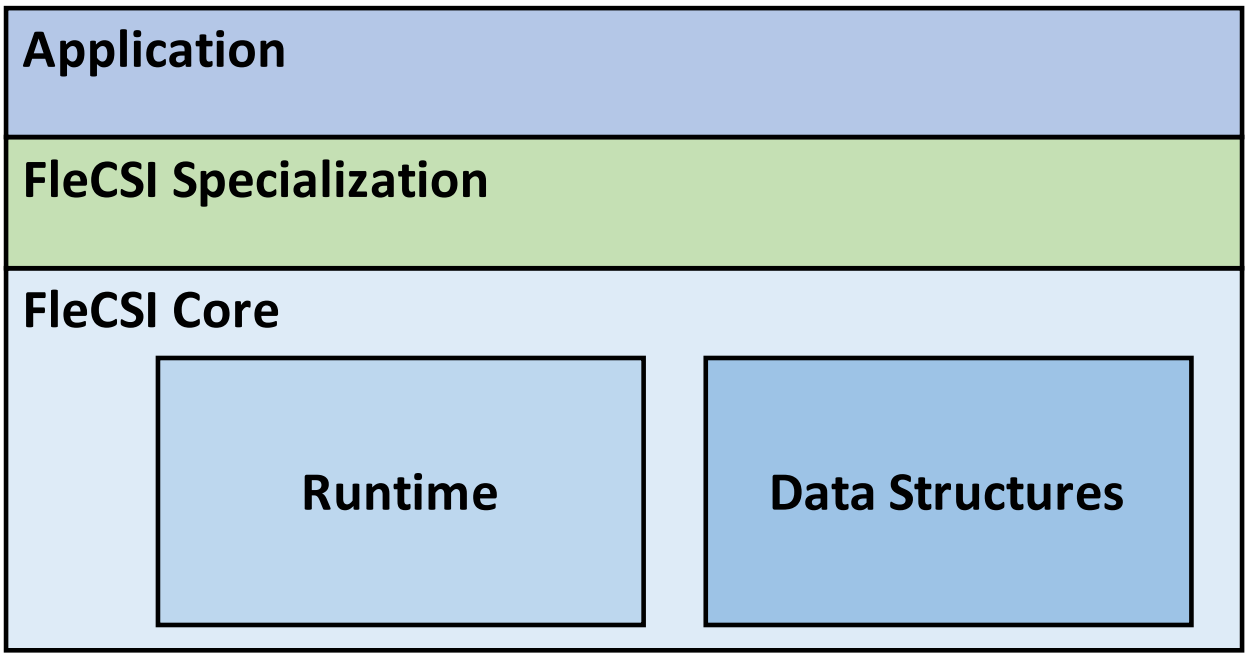

Figure 1:

- **execution**
  Execution model types.
- **topology**
  Topology types.
- **io**
  I/O types.
- **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 }.
```

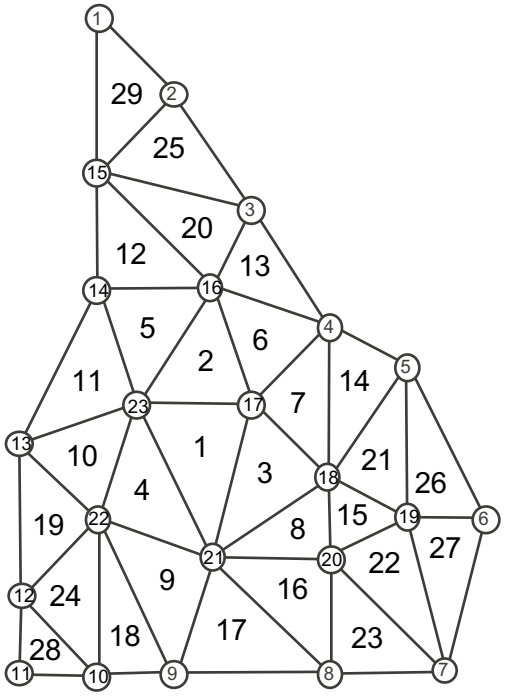

Figure 2: A canonical triangle mesh example with cells and vertices.

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 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 }
```

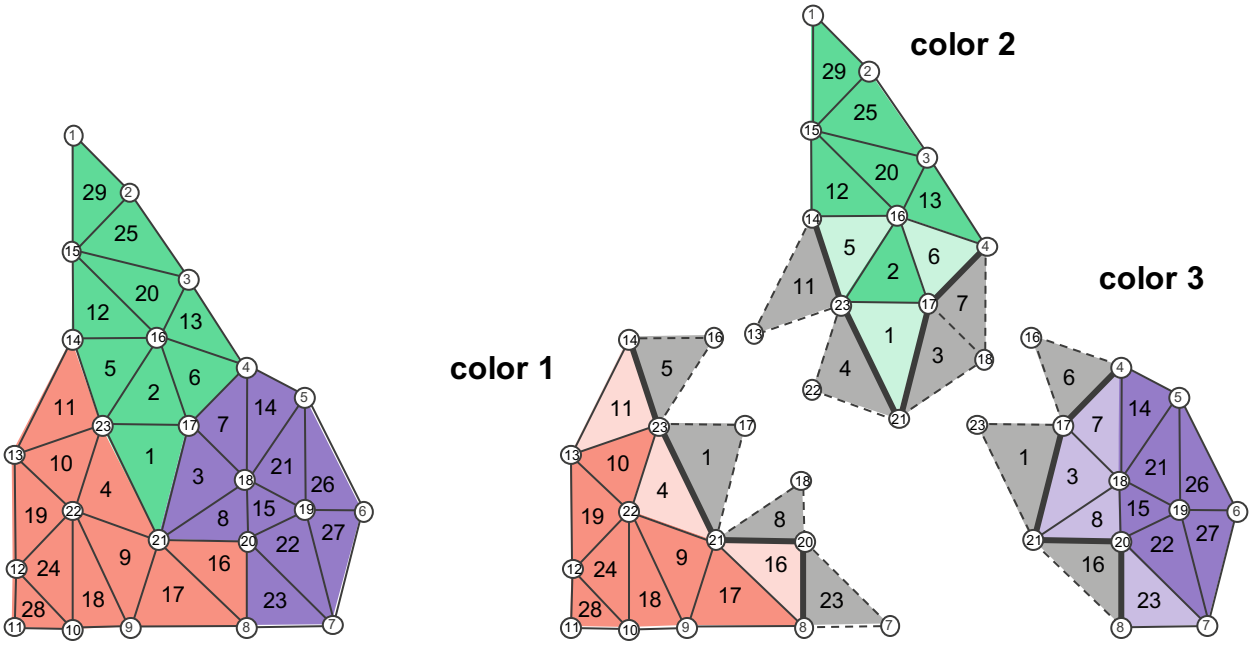

Figure 3: Disjoint coloring of our canonical triangle mesh.

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 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 $D_1 \rightarrow D_2$, transpose creates connectivity $D_2 \rightarrow D_1$. Intersect, given $D_1 \rightarrow D'$ and $D' \rightarrow D_2$, computes $D_1 \rightarrow D_2$.

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

Utilities

The utils namespace provides...

Basic Utilities

FIXME: Just list what is available. The real documentation for this is going to be in the Doxygen documentation.

Design-By-Contract Assertions (DBC)

Template Metaprogramming

Tuple Manipulation

Static Verification

Static Container

FIXME: Make sure to explain Argument-Dependent Lookup (ADL)

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.

Style Guide

If not otherwise indicated, the FleCSI coding style follows the Google C++ Style Guide.

Notable exceptions include:

- C++ Exception Handling
- Type Names
- Function Names
- Variable Names (in some cases)
- Structs vs. Classes
- Formatting

The exceptions are covered in the following sections.

Guiding Principles

- No line in a file shall exceed 80 characters!
- If you are editing a file, maintain the original formatting unless it violates our style guide. If it does, fix it!
- For the most part, all names are lowercase and follow the conventions of the C++ Standard Template Library.
- All delimiters should be terminated with a C++-style comment:

  ```
  struct trivial_t {
    double value;
  }; // struct trivial_t <- This is the delimiter comment
  ```

- Conditional and loop logic should use explicit delimiters:

  ```
  for(size_t i{0}; i<10; ++i) do_it(i); // WRONG!
  ```

  Correct way:

  ```
  for(size_t i{0}; i<10; ++i) {
    do_it(i);
  } // for
  ```

- FleCSI header includes should use the full relative path from the top-level FleCSI source directory, e.g.:

  ```
  #include "../mesh_topology.h" // WRONG!
  ```

  Correct way:

```
#include <flecsi/topology/mesh_topology.h>
```

- FleCSI header guard names should use the partial relative path. They should be lower case, and they should be appended with an underscore h (_h). The endif statement should be appended with a C++-style comment repeating the guard name. As an example, if the header file is in **'flecsi/partition/dcrs.h'**, its guard entry should look like:

```
#ifndef HEADER_HH // WRONG!
```

Correct way:

```
#ifndef partition_dcrs_h
#define partition_dcrs_h

// Code...

#endif // partition_dcrs_h
```

Directory Structure

The source code for the core FleCSI infrastructure is located in the *top-level/flecsi* directory. For the most part, the subdirectories of this directory correspond to the different namespaces in the core infrastructure. Each of these subdirectories must contain a valid CMakeLists.txt file. However, none of their children should have a CMakeLists.txt file, i.e., the build system will not recurse beyond the first level of subdirectories. Developers should use relative paths within a CMakeLists.txt file to identify source in subdirectories.

Unit test files should be placed in the *test* subdirectory of each namespace subdirectory. By convention, developers should not create subdirectories within the test subdirectory.

Names and Order of Includes

This is **not** an exception to the Google C++ Syle Guide! Please read the guide and follow its conventions.

Struct & Class Conventions

This section describes the basic conventions used in defining structs and classes. For some examples of correctly formatted type definitions, please look at Appendix A.

Structs vs. Classes

The public interface should appear at the top of the type definition when possible.

According to many sources, developers should prefer *class* to *struct*. The only real difference between the two definitions is the default access permissions, i.e., *struct* defaults to public, and *class* defaults to private.

For FleCSI, we mostly follow the Google C++ Style Guide, which prefers *class* over *struct* unless the type is intended to offer direct access to its data members. An exception to this rule is for metaprogramming types. Many of the types used in metaprogramming do not have any data members (they only provide type definitions). In this case, developers should prefer *struct* over *class*.

Like the Google C++ Style Guide convention, developers should always use a struct for type definitions that do not have restricted access permissions.

Variable Names

This is *mostly* not an exception to the Google C++ Style Guide, so you should read the guide and understand its conventions for variable names. In FleCSI, we follow those conventions for classes, and for structs that do not have restricted access permissions. For structs that **do** have access permissions, we follow the Google C++ Style Guide convention for classes.

Formatting

Control Flow

Control flow operations should not insert spaces:

```
for ( size_t i{0}; i<N; ++i ) { // WRONG!
} // for

if ( condition ) {} // WRONG!
```

Correct way:

```
for(size_t i{0}; i<N; ++i) {
} // for

if(condition) {
} // if
```

Braced Initialization

Braced initialization *should* use spaces:

```
std::vector<size_t> vec = {1,2,3}; // WRONG!
```

Correct way:

```
std::vector<size_t> vec = { 1, 2, 3 };
```

Function & Method Formatting

Function and method invocations should not insert spaces:

```
my_function ( argument1, "argument 2" ); // WRONG!
```

Correct way:

```
my_function(argument1, "argument 2");
```

More on spaces...

Never put empty characters at the end of a line!

Function & Method Formatting (prototypes)

Functions and methods should be formatted with each template parameter, the scope (static, inline), the return type, the name, and each signature parameter on its own line:

```
template<
  typename TYPENAME1,
  typename TYPENAME2,
  typename TYPENAME3
>
static
return_t &
name(
  argument1 arg1name,
  argument2 arg2name,
  argument3 arg3name
)
{
} // name
```

Parameters should have one tab equivalent indentation. The convention is to define a tab as two spaces. FleCSI source files have formatting hints for Vim and Emacs to expand tabs to this number of spaces.

NOTE: If the parameters to a function or method definition are trivial, i.e., there is only a single template parameter, **or** there are no signature parameters, it is not necessary to break up the arguments:

```
// Trivial template and signature
template<typename TYPENAME>
return_t &
name()
{
} // name

// Trivial template
template<typename TYPENAME>
return_t &
name(
  argument1 arg1name,
  argument2 arg2name
)
{
} // name

// Trivial signature
template<
  typename TYPENAME1,
  typename TYPENAME2
>
return_t &
name()
{
} // name
```

Type Names

FleCSI follows a C-style naming convention of all lower-case letters with underscores. Fully-qualified types should also append an underscore lower-case *t*, i.e., _t to the end of the type name:

```
struct my_type_t
{
  double value;
}; // struct my_type_t
```

Type definitions should be terminated with a C-style comment indicating the type name.

Template Type Naming

For templated types, use a double underscore for the unqualified type:

```
my_template_type__
```

This allows the type to be fully qualified using the normal type naming convention listed above, e.g.:

```
// Unqualified type definition
template<typename TYPENAME>
struct my_template_type__
{
  TYPENAME value;
}; // struct my_template_type__

// Fully qualified type
using my_template_type_t = my_template_type__<double>;
```

The double underscore was chosen so that it does not conflict with member variable names, which use a single underscore.

Template Parameter Names

Template parameters should use descriptive, uppercase names:

```
//----------------------------------------------------------------------//
//! @tparam TYPENAME The POD type.
//! @tparam ARGUMENTS A variadic list of arguments.
//----------------------------------------------------------------------//

template<typename TYPENAME, typename ... ARGUMENTS>
```

Error & Exception Handling

Use assertions and static assertions to assert things that must be true:

```
template<size_t FROM_DIMENSION>
connectivity &
get(
  size_t to_dim
)
{
```

```
  static_assert(FROM_DIMENSION <= DIMENSION, "invalid from dimension");
  assert(to_dim <= DIMENSION && "invalid to dimension");
  return conns_[FROM_DIMENSION][to_dim];
} // get
```

In this case, we can verify that the from dimension (template parameter FROM_DIMENSION) is in bounds using a static assertion. We need to use a dynamic assertion to check the to dimension (to_dim) since it is passed as an argument to the method. In both cases, the assertions must be true for the code to not be broken, i.e., if the assertion is not true, there is a bug! Fix it!

Use exception handling to catch exceptional situations, i.e., when a condition for the correct functioning of the code is not met. An exception may be caught and the program can recover from it:

```
try {
  type_t * t = new type_t;
}
catch(std::bad_alloc & e) {
  // do something because the allocation failed...
}
catch(...) {
  // default exception
} // try
```

In many cases, exception handling should be reserved for interfaces that can be called by a developer. Internal interfaces should use assertions to identify bugs.

Summary

Failure to respect the FleCSI style guidelines will lead to public ritualized torture and eventual sacrifice...

Appendix A: Style Examples

```
//----------------------------------------------------------------------//
//! The my_interface_t type provides an example of a correctly formatted
//! and documented type.
//----------------------------------------------------------------------//

template<
  typename TYPENAME
>
struct my_interface_t
{

  //----------------------------------------------------------------------//
  //! Construct a my_interface_t with value.
  //----------------------------------------------------------------------//

  my_interface_t(
    TYPENAME value
  )
  :
    value_(initialize(value))
  {}

  //----------------------------------------------------------------------//
  //! This method provides an example of a member function.
  //!
  //! @param input The input value to the method.
  //!
  //! @return A modified value of type TYPENAME.
  //----------------------------------------------------------------------//

  TYPENAME
  two_times(
    TYPENAME input
  )
  {
    return 2.0*value_;
  } // two_times

private:

  // private member functions

  void
  initialize(
    TYPENAME value
  )
  {
    return value + 5.0;
  } // initialize
```

```
  // private data members

  TYPENAME value_;

}; // struct my_interface_t
```

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.

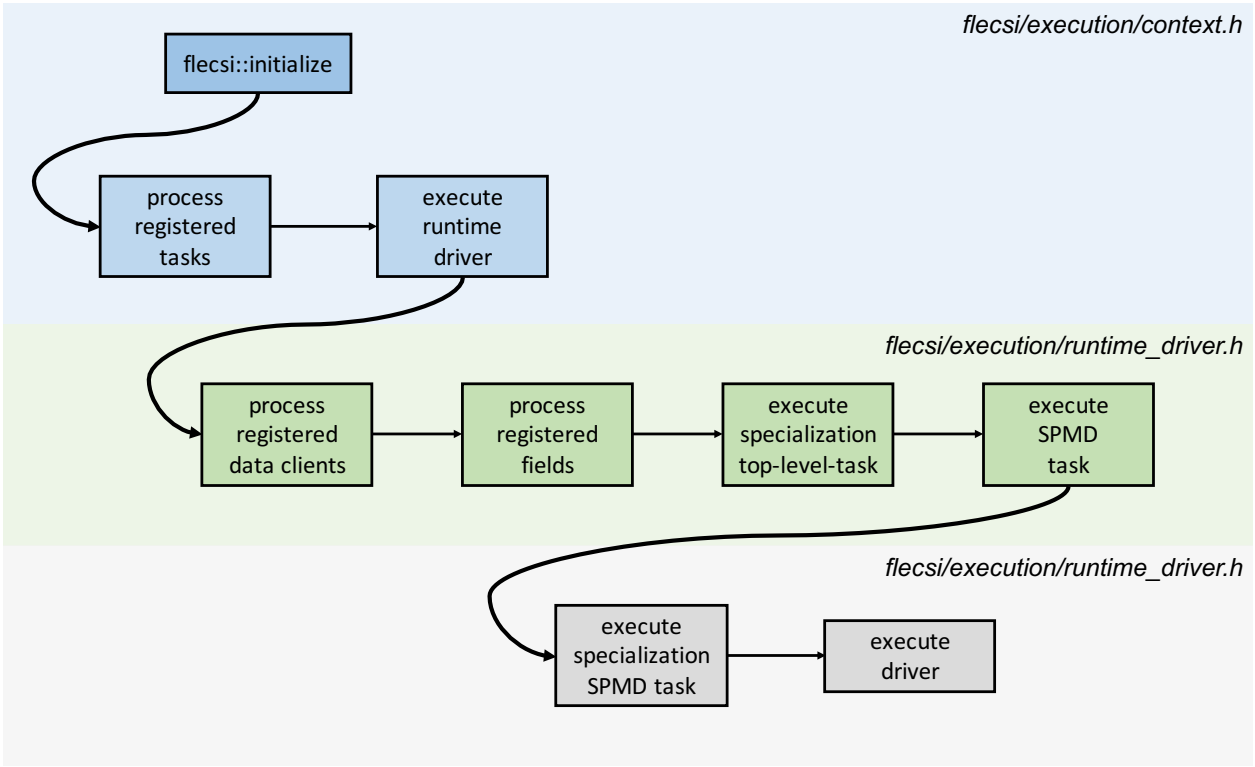

Figure 4: FleCSI Runtime Initialization Structure.

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:

```
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.

Writing Specializations for FleCSI

- associate ids with index spaces
- document required types and interfaces for different topologies

A channel to allow communication between FleCSI users and developers to speed up responses to problems in building and using FleCSI.

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.