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

Tim Kelley
Applied Computer Science (CCS-7)
Los Alamos National Laboratory
tkelley@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

Josh Payne
Applied Computer Science (CCS-7)
Los Alamos National Laboratory
payne@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
Amy Hungerford
Nathaniel Morgan
Joe Schmidt
John Wohlbier

Contents

Introduction **3**

Name Spaces **3**

Unit Tests **4**

Data Model **4**

Execution Model **4**
- Tasks . . . 5
- Functions . . . 5
- Kernels . . . 5
- FleCSIT . . . 5

Mesh Coloring **5**

FIXME: Working notes for structured mesh interface **5**

Topology Types **6**
- Mesh Topology . . . 6
- N-Tree Topology . . . 6
- K-D Tree Topology . . . 7

I/O **7**

Utilities **7**
- Basic Utilities . . . 7
- Design-By-Contract Assertions (DBC) . . . 7
- Template Metaprogramming . . . 7
  - Tuple Manipulation . . . 7
  - Static Verification . . . 7
  - Static Container . . . 7

Style Guide **7**
- Guiding Principles . . . 8
- Directory Structure . . . 9
- Names and Order of Includes . . . 9
- Struct & Class Conventions . . . 9
  - Structs vs. Classes . . . 9
- Variable Names . . . 9
- Formatting . . . 9
  - Control Flow . . . 9
  - Braced Initialization . . . 10
  - Function & Method Formatting . . . 10
  - More on spaces... . . . 10
  - Function & Method Formatting (prototypes) . . . 10
- Type Names . . . 11
  - Template Type Naming . . . 11
  - Template Parameter Names . . . 12
- Error & Exception Handling . . . 12
- Summary . . . 13

Appendix A: Style Examples **14**

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.

Name Spaces

FleCSI uses several different namespaces:

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

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.

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 accessors, 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.
- **local**
  This storage type is designed to provide scratch space that does not need to be managed by the runtime.
- **tuple**
  This storage type provides c-struct-like support so that task arguments can be associated without forcing a particular data layout.

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 a serial runtime, and 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 serial implementation is in *serial*. 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, i.e., pure functions with controlled side-effects.

Functions

Kernels

FleCSIT

FleCSI offers two primary styles of development: The *FleCSIT* compilation tool that allows fast prototyping of multi-physics ideas, and the application interface that is intended for production code development.

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.

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)

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 vec = {1,2,3}; // WRONG!

Correct way:

std::vector 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 T1,
  typename T2,
  typename T3
>
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 T>
return_t &
name()
{
} // name

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

// Trivial signature
template<
  typename T1,
  typename T2
>
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 T>
struct my_template_type__
{
  T 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 single letter, uppercase names unless they are variadic, in which case, a lowercase *s* should be appended to the name:

```
/*!
  \tparam T The POD type.
  \tparam As A variadic list of arguments.
 */
template<typename T, typename ... As>
```

In this example, the *s* in *As* indicates that the parameter is plural. Developers should avoid verbose parameter names, opting instead to use Doxygen to document the parameter meaning (as shown above).

Error & Exception Handling

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

```
template<size_t FD>
connectivity &
get(
  size_t to_dim
)
{
  static_assert(FD <= D, "invalid from dimension");
  assert(to_dim <= D && "invalid to dimension");
  return conns_[FD][to_dim];
} // get
```

In this case, we can verify that the from dimension (template parameter FD) 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

```
/*!
  \struct my_interface_t my_interface.h
  \brief my_interface_t provides an example of a correctly formatted
    and documented type.
 */
template<typename T>
struct my_interface_t
{

  /*!
    Constructor.

    \param value Initialization value...
   */
  my_interface_t(T value) : value_(initialize(value)) {}

  /*--------------------------------------------------------------------------*
   * Section delimiter.
   *--------------------------------------------------------------------------*/

  /*!
    This method provides an example of a member function.

    \param input The input value to the method.

    \return A modified value of type T.
   */
  T
  two_times(
    T input
  )
  {
    return 2.0*value_;
  } // two_times

private:

  // private member functions

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

  // private data members

  T value_;
```

```
}; // struct my_interface_t
```
