Contents

Introduction

FleCSI is a set of tools that provide mid- and low-leve interfaces that can be used to create high-level abstractions and interfaces that are suitable for computational physicists and computational scientists.

This Guide...

The Developer Guide

Doxygen Documentation

I/O

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.)

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.

Data Model

From the user's point of view, the FleCSI data model is extremely easy to use. Users can register data of any normal C++ type. This includes P.O.D. (plain-old-data) types and user-defined types.

Data Clients

Field Data

Handles

Permissions

In order for the FleCSI runtime to infer distributed-memory data dependencies that must be updated during task execution, the user must specify what permissions are required for each client or field handle type for each task. Currently, permissions are specified as template-style arguments to the parameters in the task signature:

```
void task(mesh<ro> m, field<rw, rw, ro> f) { // ...
```

The permissions specifiers have the following meanings:

- **ro** [**read-only**]
  Read-only access implies that ghost values will be updated if necessary before the user task is executed.
- **wo** [**write-only**]
  Write-only access implies that shared values will be updated if necessary after the user task is executed.
- **rw** [**read-write**]
  Read-write access implies that ghost values will be updated if necessary before the user task is executed, and that the shared values will be updated if necessary after the user tash is executed.

Storage Classes

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.

Example

```
flecsi_register_data_client(mesh_t, hydro, m);
flecsi_register_field(mesh_t, hydro, pressure, double, dense,
  1, cells);

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {

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

} // update_pressure
```

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.

Kitsune Project

The FleCSI C++ language extensions are implemented as part of the Kitsune project. Kitsune provides support for compiler-aided tools development:

- **Static Analysis & Optimization**

Kitsune provides an interface for defining AST visitors that recognize user-defined regular expressions and perform static analysis or supercede normal compilation to optimize specific code patterns.

- **Domain-Specific Language (DSL) Development**

Kitsune also provides an interface for defining custom language extensions.

Access to Kitsune is currently restricted to yellow-network users at LANL. If you need access, please contact Pat McCormick (pat@lanl.gov).

Once you have access to the repository, you can obtain the source and build like:

```
$ git clone git@gitlab.lanl.gov:pat/kitsune.git
$ cd kitsune
$ mkdir build
$ cmake -DCMAKE_BUILD_TYPE=Release -DLLVM_TARGETS_TO_BUILD=X86 -DCMAKE_INSTALL_PREFIX=/path/to/install .
$ make install
```

If you are a developer, you may want to specify *Debug* for the build type.

Kitsune is the Japanese word for *fox*.

FleCSI C++ Language Extensions

Currently, FleCSI provides only the *forall* keyword extension to C++ for defining fine-grained, data-parallel operations on logically shared data regions.

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

FleCSIT Tool

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.