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
}
```