research note

Computer and Computational Sciences Division

CCS–7:Applied Computer Science

To/MS: File

From/MS: FleCSI Design Team:
Ben Bergen, CCS–7, MS B287
Marc Charest, XCP–1, MS P365
Irina Demeshko, BCCS–7, MS 287
Timothy Kelley, CCS–7, B287
Nick Moss, CCS–7, MS B287
Josh Payne, CCS–7, MS B287
John Wohlbier, CCS–2, MS D413
Geoff Womeldorf, CCS–7, MS B287

Phone/FAX: 7–3965

Symbol: CCS-7-xyz

Date: March 24, 2016

Subject: FleCSI Data Management Requirements
Revision: DRAFT 1.0

Contents

1 Introduction

The goal of this Research Note is to identify requirements for the FleCSI data model. Additional documentation will discuss design options, testing, and so on.

There are a number of different areas of data management:

1. data associated with mesh elements (so-called *mesh fields*);
2. particle data;
3. data associated with physical models (equation of state, for example);
4. simulation state data that is set at launch time, but does not change afterward (physics configuration, for example);
5. simulation state data that varies during the run (cycle number, for instance).

A field can be thought of as a relation from a set of mesh entities (such as cell centers or edges) to some other set of objects (such as double precision numbers or geometric vectors). It is useful to distinguish between the Field, which is the abstract object or mapping, and the FieldData, the actual velocity in cells 1,2,3, etc. Field and FieldData requirements are discussed in §2.

Physical model data is (typically) in tabular form, and represents the results of (usually) expensive constitutive modeling. The use cases and requirements for physical model data are covered in §4.

Static and mutable simulation data include control data, as well as physical and computational configuration data. These use cases and requirements are covered in §5 (constant) and §6 (mutable).

1.1 FleCSI Programming Model

The computing model that we have in mind features the elements summarized in Table 1.

Processing Element	Description
Application	(Runtime?): instantiates DataStore and objects that interact with it;
DataStore	manages fields, field data (as well as other types of data not directly associated with mesh elements)
Package	namespace for tasks, typically associated with a physics operator, provides Tasks and one or more Drivers;
Driver	Shares data between tasks and , can interact with the DataStore to coordinate
Task	Coordinates kernels to accomplish a task. Can interact with the DataStore, can operate on field data passed to it by its Driver; tasks can interact with the data store to obtain temporary state, make state visible to the application.
Kernel	stateless computation, it operates on the data provided to it by its task; kernels do not interact with DataStore.

Table 1: Actors in the FleCSI computing model.

1.1.1 Kernel

A kernel is pure function that accomplishes a single, well–defined job. In physics terms, a kernel computes some interesting quantity from its inputs and returns that quantity as its output. At first, think of a kernel as code to compute a single thing—that is, one number. The application of the kernel over a larger collection, such as a data field, will be performed by something else.

Pure in this context means that the invocation of the kernel can be replaced by the return value of the kernel. This means that the kernel cannot do many things. A kernel cannot communicate; it cannot print something to the screen; it cannot store data that would affect future invocations. It's meant to be a function in a very mathematical sense: it's a machine whose behavior is completely dictated by its inputs.

Note that this does not preclude a kernel from being a callable object (in one of C++'s more egregious parochialisms, a "functor"). It just means that invoking the kernel cannot change the state of the object. Thus, one could parameterize the kernel object at construction time, then leave it alone. Syntactically, this is equivalent to declaring the call operator const, for example:

```
struct Kernel58
{
    ...
    ReturnT operator()(InputT i, ...) const {...}
    ...
};
```

1.1.2 Task

A Task coordinates one or more Kernels to accomplish a more complicated undertaking. For example, a Task might apply a kernel to each element of a Field. A larger scale Task would coordinate multiple kernels.

1.1.3 Driver

A driver is a task that holds special status as one of a few principle entry points of a package.

1.2 Data Policies for FleCSI Programming Model

Policy choices will create requirements for the various elements that will need to be reflected in the design. Core policy decisions include access and lifecycle.

Policies also need to distinguish between the various roles that tasks play. For example, data mutating tasks need to be very careful about access to data. Non-mutating tasks, such as monitoring, can probably be given freer access.

Programmers also need to be careful about data access. Accessing data from the DataStore essentially limits the reusability of a piece of code. Such accesses should be carefully encapsulated.

1.2.1 Possible Policy 1: 'The-Hell-With-It'

One policy is everyone gets everything at any time. The advantage of this approach is expedience: it requires no thought. The disadvantage of this approach is that in effect, it turns the DataStore into a colossal common block. Though a historically popular choice, The-Hell-With-It policy is maintenace nightmare, and it fundamentally obscures the physics protocol (operator coupling). This policy would definitely be a Last Generation Code choice.

1.2.2 Possible Policy 2: Limited Hierarchy Policy

Another policy would seek to enforce the usual hierarhical approach to program organization. In this approach, data objects can only be passed downward, one level at a time.

In more detail, the rules of this policy are:

1. no data is passed (taken) laterally,
2. passing data between levels of the hierarchy must be explicit,
3. that lower elements use data objects provided by the higher levels,
4. that higher elements do not access objects created by lower elements,
5. and that lower elements do not use data objects from siblings.

Refering to the diagram in Fig. 1, we could immediately infer the following constraints:

1. T3 can use read/write data provided by T1;
2. T3 cannot use any data created by T2, nor any data provided by T1 to T2, unless T1 happens to provide that data to T3 as well;

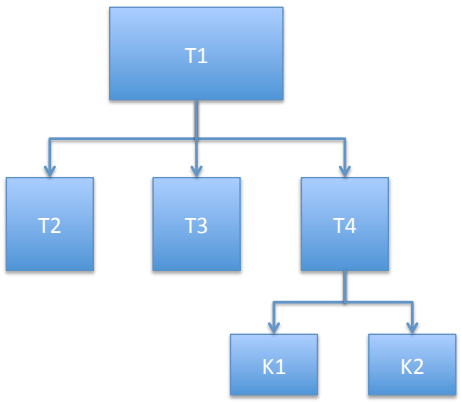

Figure 1: A simple processing network.

3. T1 could not use data created by T3;
4. K2 could use data provided by T4;
5. The only way that K2 could use data provided by T1 would be if T4 explicitly relayed it to K2.

2 Field Requirements

2.1 Use Cases

Use Case	Description
U1.1	adds a field on request: adds metadata, associates metadata to field
U1.2	allocates the FieldData corresponding to the field in the right memory
U1.3	provides accessors to field data
U1.4	deletes a field
U1.5	deletes FeildData
U1.6	retrieves a field
U1.7	searches fields for matching metadata

Table 2: Use cases for DataStore

Use Case	Description
U2.1	Is given read or write access to a field
U2.2	updates a field
U2.3	can declare or initialize a field
U2.4	associates a field with an index set (dense or sparse)

Table 3: Task/Driver use cases for Fields.

2.2 Field Requirements

Table 4: Requirements for Fields

Requirement	Description
	Metadata
R1.1	Associate a name with every Field. *Discussion: Does the name need to be unique? Currently, the name is a unique identifier. It's not clear that that needs to be the case, though. For example, can there be a density field for all materials combined (hydro view), and a density field associated with chocolate, peanut butter, etc (chemistry view)? But if there are multiple fields with the same name, how does two packages refer to the same field? Namespacing?*
R1.2	Associate a namespace with every Field. *Note 1: The namespace can be defaulted.* *Note 2: A name can occur zero or once in a namespace.*
R1.3	Associate a version with each field. *Note 1: This may be required for techniques like predictor-corrector methods.* *Note 2: The version distinguishes between multiple instances of the same field name.* *Example: 'pressure after time step 42' versus 'pressure after time step 43'.* *Discussion: Name, namespace, and version uniquely identify a field.*
R1.4	Associate a unique integer identifier with each field. *Example: We want to be able to talk about a data object independently of its name.* *Note 1: an id would be more convenient than specifying name, namespace, and version.*
R1.5	Associate Fields with various builtin metadata aspects. *Note 1: a field can be associated with 0 or more materials, 0 or more isotopes.*
R1.6	Permit users to register metadata in the form of <Key,Value> pairs. *Note 1: Also permit users to search for and set user–define <Key,Value> metadata.*
R1.7	Search for Fields matching a given metadata criterion.
R1.8	Set each metadata attribute for each Field.
	Topological Elements
R1.9	Associate Fields with homogeneous sets of elements of one mesh. *Note 1: "homogeneous": only one type of element in the set.* *Note 2: a field may be associated with any subset of the elements of a mesh, including all of the elements of that type on the mesh;* *Note 3: a field need only be associated with the elements of one mesh.* *Note 4: Arbitrary mesh entities, including cell centers, edges, faces, vertices, corners, wedges.*
	Data Elements
R1.10	Associate fields with FieldData, i.e. the actual data. *Note 1: There is a bijective relationship between Fields and FieldDatas.* *Note 2: The type of FieldData is fairly arbitrary; it probably includes any type with a null constructor—any type that can be allocated via a new [] statement. Further specification is required.*
R1.11	Given a field descriptor, get read and write access to the FieldData. *Note 1: Read access may be separated from write access.* *Note 2: It would be nice to have a way of saying "done writing this field".*

Table 4: (cont.) Requirements for Fields

Requirement	Description
R1.12	Associate each Field with an IndexSet. *Note 1: Exactly one IndexSet per Field.*

Another key concept is the index set with which Field is associated. The index set defines the subset of mesh entities to which the FieldData correspond.

2.3 FieldData

FieldData are the actual variables associated with the mesh elements. They represent physically or computationally interesting quantities. The representation of these data is closely bound up with effecient iteration over them. Requirements on these data need to be sensitive to the possibility of needing different views and different layouts of the data for different machines and different algorithms.

Requirement	Description
R2.1	This table should capture requirements for FieldData.

Table 5: Requirements for accessing FieldData, getting to it, allocation it, initializing and persisting it.

3 Particle Requirements

Fields acquire strcuture naturally from the (sub)meshes with which they are associated. Particles are not always so easily structured. This requires care and maintenance in specifying requirements for particles. We assume there will be some kind of organizing principle above the level of individual particles. Thus we distinguish between:

1. ParticleSet, a descriptor of a set of particles,
2. ParticleData, space for recording the attributes of particles,
3. ParticleIndex, a "virtual" grouping of particles.

3.1 Use Cases

3.2 ParticleSet Requirements

3.3 ParticleData

4 Physical Model Data Requirements

Physical models are used to represent physics that is too expensive to compute on–the–fly. Instead, that data is typically stored in tabular form. Many tables may be required in the course of a simulation. This section captures the requirements for such data.

Use Case	Description
U3.1	adds a ParticleSet on request: adds metadata, associates metadata to particle
U3.2	allocates the ParticleData corresponding to the particle in the right memory
U3.3	provides accessors to particle data
U3.4	deletes a particle set
U3.5	deletes particle data
U3.6	retrieves a ParticleSet
U3.7	retrieves a ParticleData
U3.8	searches ParticleSets for matching metadata

Table 6: DataStore use cases for particles.

Use Case	Description
U4.1	Is given read or write access to ParticleData
U4.2	updates ParticleData
U4.3	can declare or initialize a ParticleSet
U4.4	can declare or initialize ParticleData
U4.5	can declare or initialize ParticleIndex

Table 7: Task/Driver use cases for particles.

5 Static Simulation Configuration Data Requirements

Certain data is specified at simulation launch time; it remains static throughout the run. This data may be required throughout the simulation. It may include physical data, such as the domain extents, or computational data, such as the starting time step, or the (static) configuration of the run. This data excludes things that change over the course of the simulation, such as current simulation time, or current mesh partition.

6 Mutable Simulation Configuration Data Requirements

TMK:tk

Requirement	Description
R3.1	This needs more specification

Table 8: Requirements for Physical Model Data

Requirement	Description
R4.1	This needs more specification

Table 9: Requirements for Simulation Launch Data

Requirement	Description
R5.1	Partition Data
R5.2	Cycle Information *Note 1: to include: cycle number, cycle start time, delta-t, ... more spec needed* *Note 2: should include data that can be registered and set by the user/package.* *Note 2a: is granularity finer than cycle/time-step required?*

Table 10: Requirements for Mutable Simulation State Data

Requirement	Description
R6.1	Communications topology *Note 1: There may be multiple partitions of a single mesh.* *Note 2: There may be multiple meshes, each partitioned differently.*

Table 11: Requirements for Partitioning Data