Functions over Meshes

Matthew Knepley and Dmitry Karpeev

Mathematics and Computer Science Division
Argonne National Laboratory

FEniCS 2006
Delft University of Technology

Outline

1. Review of Mesh
2. Section Interface
3. Completion
4. Conclusions

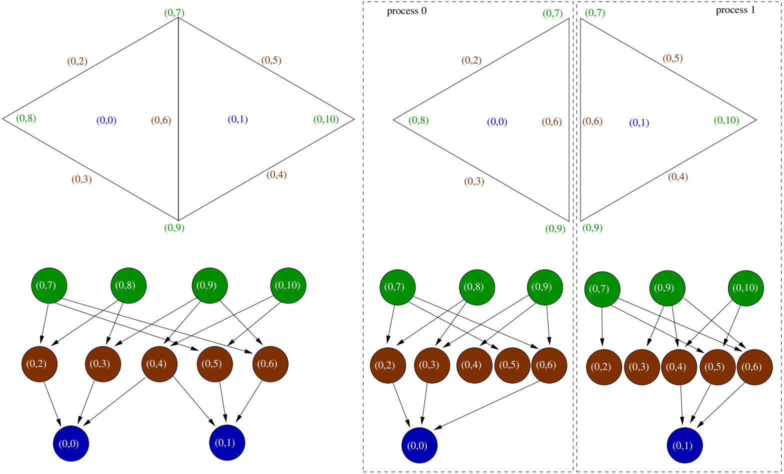
Part I

Review of Mesh

A *Topology* is a collection of points with a covering relation

- Points represent vertices, edge, . . .
- Covering relation is represented by directed edges
  - This produces a graph, called a *Sieve*
  - For meshes, the graph is DAG and is stratified
- In a `Topology`, we allow multiple `Sieves`

Sieve: Distributed Mesh



Bundle

A *Bundle* is an association of spaces to points

A *Section* is a function over these spaces

- A `Bundle` combines `Sections` with a `Topology`
- A `Mesh` is a `Bundle` over the computational topology
  - It has a distinguished Section, *coordinates*
  - The intrinsic dimension is the height/depth of the Sieve

Part II

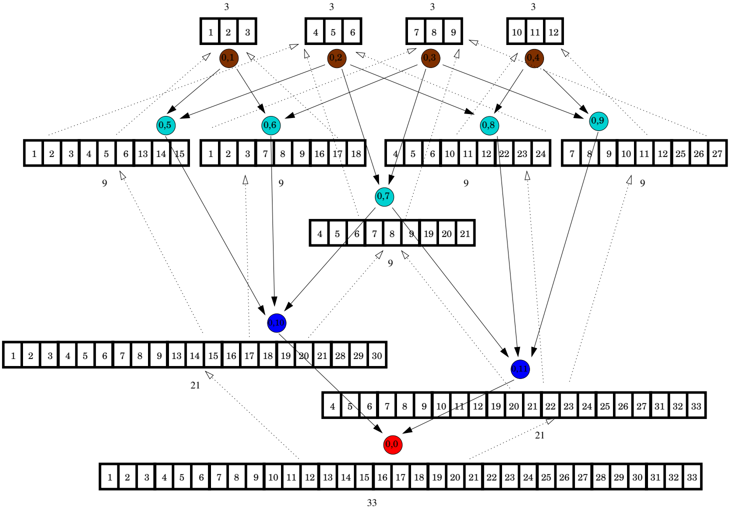
Section Interface

Section Interface

A `Section` is a mapping from Sieve points to a vector of values

- `restrict`, `update`
  - Defined on the `closure` of a point
- Use an *Atlas* to manage dimension of each fiber
  - Can be implemented by a `Section`
  - Must also manage the domain (harder)
- Participate in *completion*
  - Communicate values over an *Overlap*

Sieve: Mesh Data



Part III

Completion

Section Completion

An *Overlap* is a Sieve associating points in different Sieves

We have four phases:

- Copy local values
- Communicate sizes
  - Notice that the size is constant
- Communicate values
- Update section with remote values

Communication

We use auxiliary objects having a Section interface.

- Use a *sizer* to allocate overlap section
  - Use an Atlas and `restrict` to the point
  - Use section interface for overlap section
  - Just the completion of the Atlas
- Use a *filler* to update overlap section
  - Use a Section and `restrict` to the point
- Communicate values in overlap sections
  - Can use an arbitrary fusion strategy, not just addition or replacement

General communication routines can enable

- Mesh partition/distribution (and unification)
- Section distribution
- Load balancing

Mesh Completion

Meshes can be reduced to sections

- Discrete topology is a section over the partitions
  - Complete this section to distribute points
- Topology is a section over the discrete topology
  - Values are *cones*, in the space of points
  - Complete this section to distribute cones
- Complete associated sections

Section Implementations

- `ConstantSection`
  - Constant value over the domain
  - No communication to complete
- `UniformSection`
  - Constant fiber dimension
  - Atlas can be a `ConstantSection`
- `Section`
  - Arbitrary fiber dimension
  - Atlas can be a `UniformSection`
- Thus we have termination of a completion recursion

Concepts vs. Types

Concepts

- Sieve
- Overlap
- Section
- Atlas
- Bundle

Types

- `Sifter`
- `Sifter`
- `ConstantSection`
- `UniformSection`
- `Mesh`

Custom Assembly

To define a given assembly, we need

- Domain definition
- Overlap Construction
- Fusion operator

This could support

- FETI-DP, BDDC
- GMG
- FMM

Part IV

Conclusions

Conclusions

- Must distinguish between Concept and Type
    - Soon to be included in C++
- Can make do with two basic objects
    - Sieve
    - Section
- This vastly simplifies algorithms
    - Most notable in communication

Part V

Ideas about Build Systems

Traditional Problems

- Global namespace
  - SCons continues this make shortcoming
- Configuration and build dependencies
  - No explicit hierarchy or dependencies
- Audit trail for configure/build information
  - When did this flag/library which broke my test get included?
- Integration of configure and build
  - Uniform, structured access to configure data
- Configuration of batch systems
- Persistence

Proposals

- Encapsulation
  - Configure data in Python objects
  - Use framework `require` to access configure objects
  - Pass in build object to make rules
- Auditing
  - Some kind of transition log for designated variables
- Configure integration
  - Simple `require()` interface to the configure DAG
- Configure extensibility
  - Configure object template
- Configure for batch systems
  - Generate and build a C executable, which runs in the queue
  - This generates `reconfigure.py` which sets options correctly
- Persistence
  - Use builtin Python persistence

Dependencies

- Configure uses the framework `require('module.name', self)`
  - Returns the requested configure object
  - Creates a DAG edge between that object and `self`
  - Could be extended to interproject dependencies?
- Build still using text BNF-style
  - Should establish a full DAG underneath (broken in recursive make)
  - Auto-dependencies?

Configure Integration

- `addDefine()`, `addSubstitution()`
    - Replicates the Autoconf interface
- `addTypedef()`, `addPrototype()`
    - Better interaction with C/C++
- `addMakeMacro()`, `addMakeRule()`
    - Structured interface to `make`
- Custom build rules
    - Determine includes, libraries, and flags directly from configure
    - Can establish implicit rules
    - Use automake-like targets

Build Rules

```
def bin_foo(maker):
  'foo: foo.c bar.h'
  return
```

```
def dylib_foo(maker):
  'libfoo: foo.c bar.c'
  return (maker.mpi.include, maker.mpi.lib, [ '-DFOO'])
```

```
def dylib_bat(maker):
  'libbat: bat.c'
  return ([], [os.path.join(maker.libDir,  'libfoo.a')], [
```

Other Features

Sometimes incidental features can greatly increase usability

- PETSc Options and configure argument parsing
- Configure help system
- Integrated version control
- `importer.py`