Theoretical Foundations

Dmitry Karpeev [1,2], Matthew Knepley [1,2], and Robert Kirby [3]

[1]Mathematics and Computer Science Division
Argonne National Laboratory

[2]Computation Institute
University of Chicago

[3]Department of Computer Science
Texas Tech University

Foundations of Finite Element Computing
Simula Research, Oslo, Norway
August 3-10, 2008

Part I

Introduction

Outline

Problems

The biggest problem in scientific computing is programmability:

- Lack of usable implementations of modern algorithms
  - Unstructured Multigrid
  - Fast Multipole Method
- Lack of comparison among classes of algorithms
  - Meshes
  - Discretizations

We should reorient thinking from

- characterizing the solution (FEM)
  - "what is the convergence rate (in h) of this finite element?"

to

- characterizing the computation (FErari)
  - "how many digits of accuracy per flop for this finite element?"

Problems

The biggest problem in scientific computing is programmability:

- Lack of widespread implementations of modern algorithms
  - Unstructured Multigrid
  - Fast Multipole Method
- Lack of comparison among classes of algorithms
  - Meshes
  - Discretizations

We should reorient thinking from

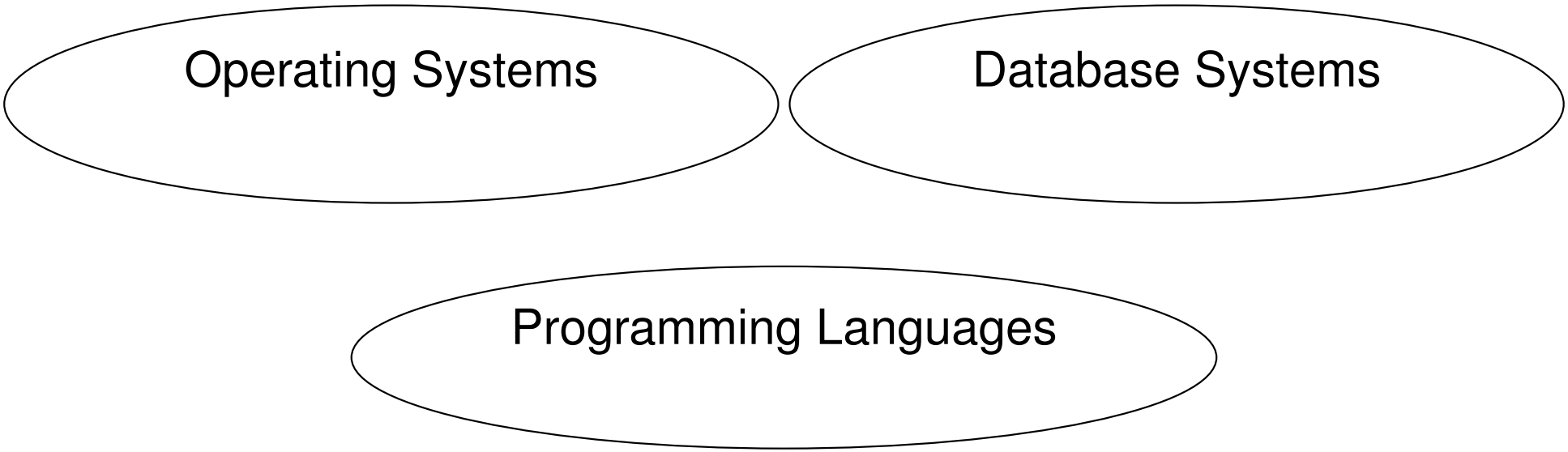
- characterizing the solution (FEM)
  - "what is the convergence rate (in h) of this finite element?"

to

- characterizing the computation (FErari)
  - "how many digits of accuracy per flop for this finite element?"

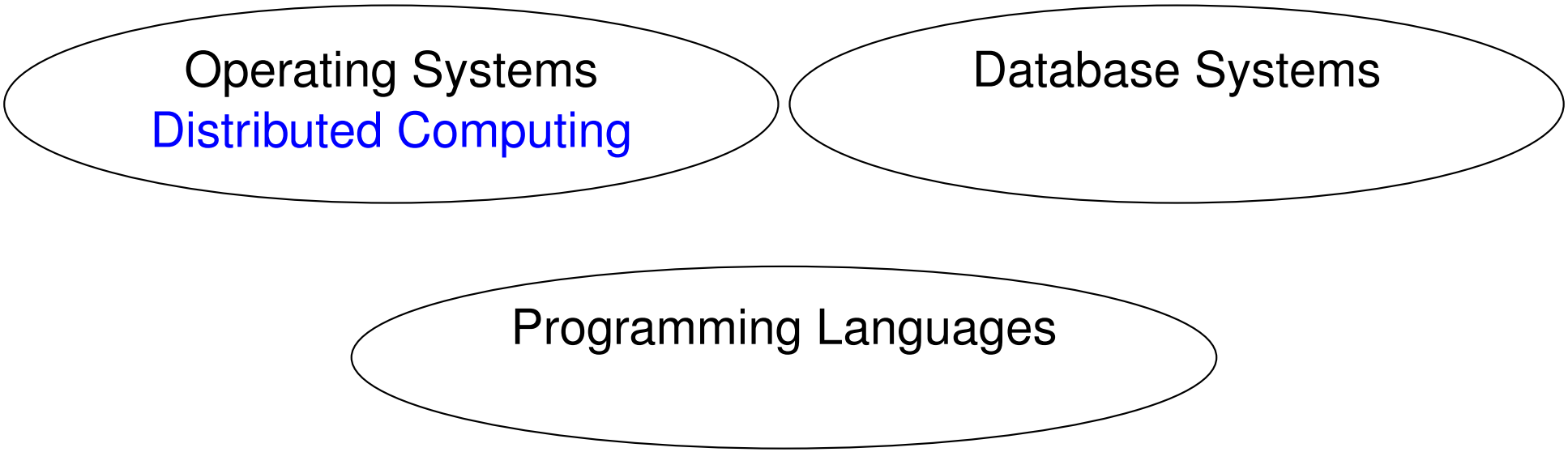
Interaction with Systems

We have to bridge the gap with Systems
to enable Scientific Computing



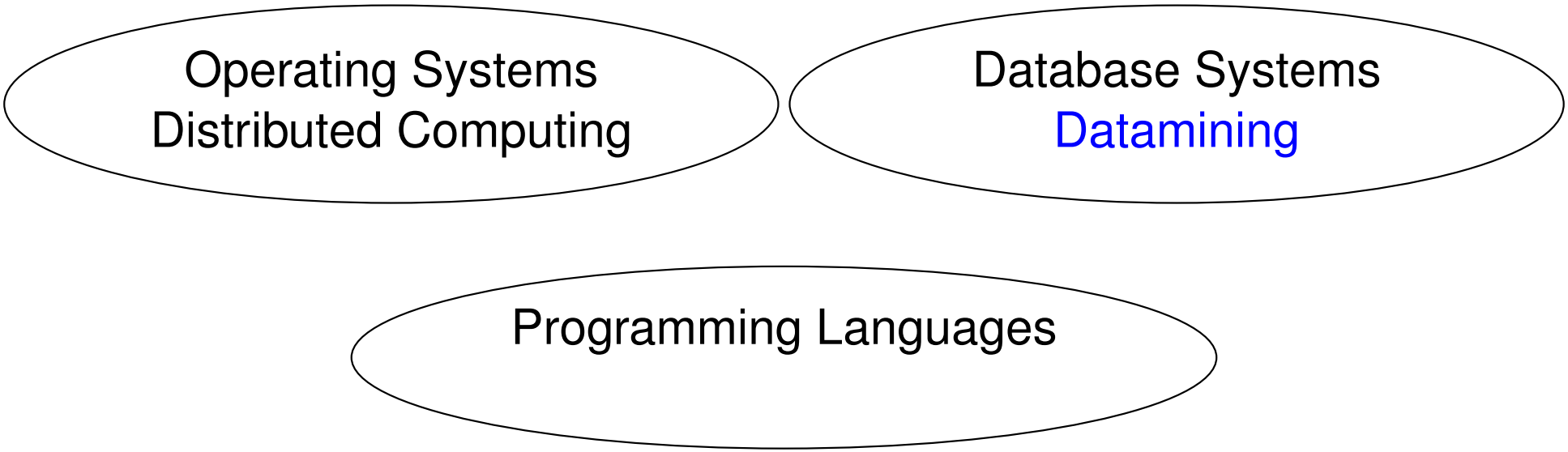
Interaction with Systems

We have to bridge the gap with Systems
to enable Scientific Computing



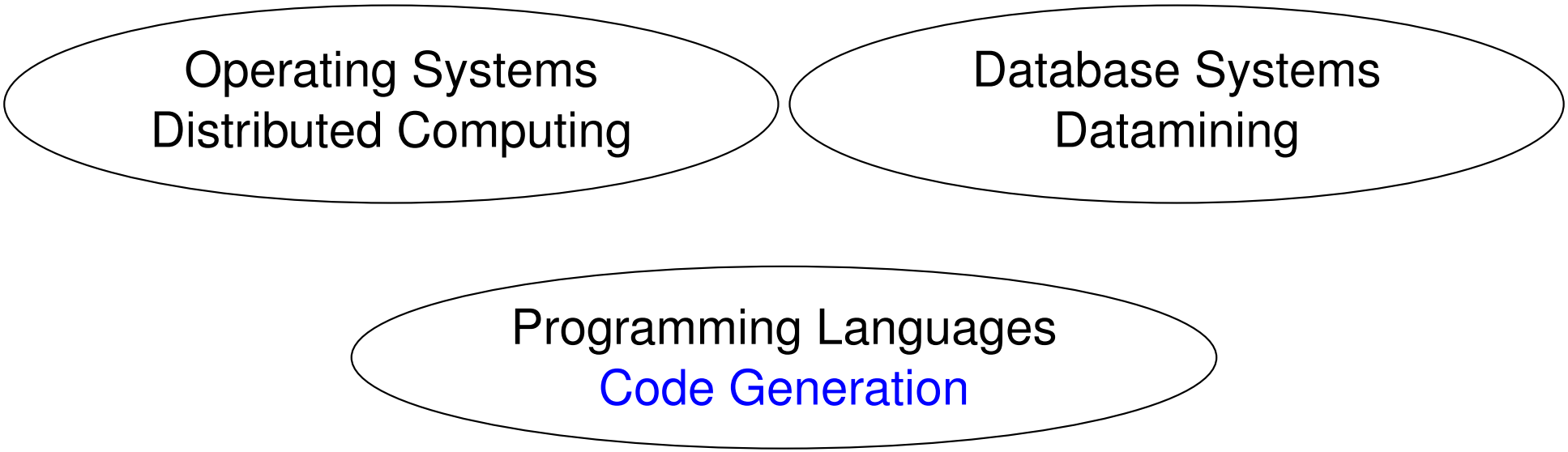
Interaction with Systems

We have to bridge the gap with Systems
to enable Scientific Computing



Interaction with Systems

We have to bridge the gap with Systems
to enable Scientific Computing

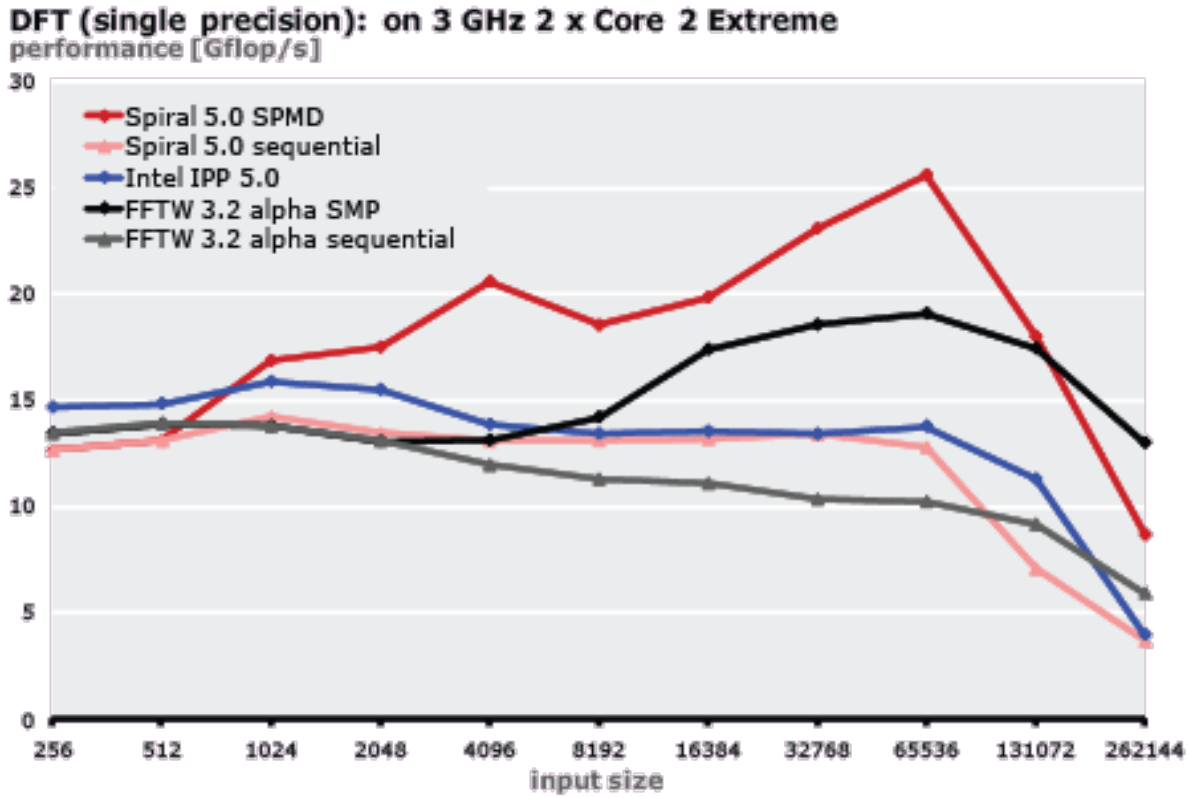


Future Compilers

I think compilers are victims of their own success (ala Rob Pike)

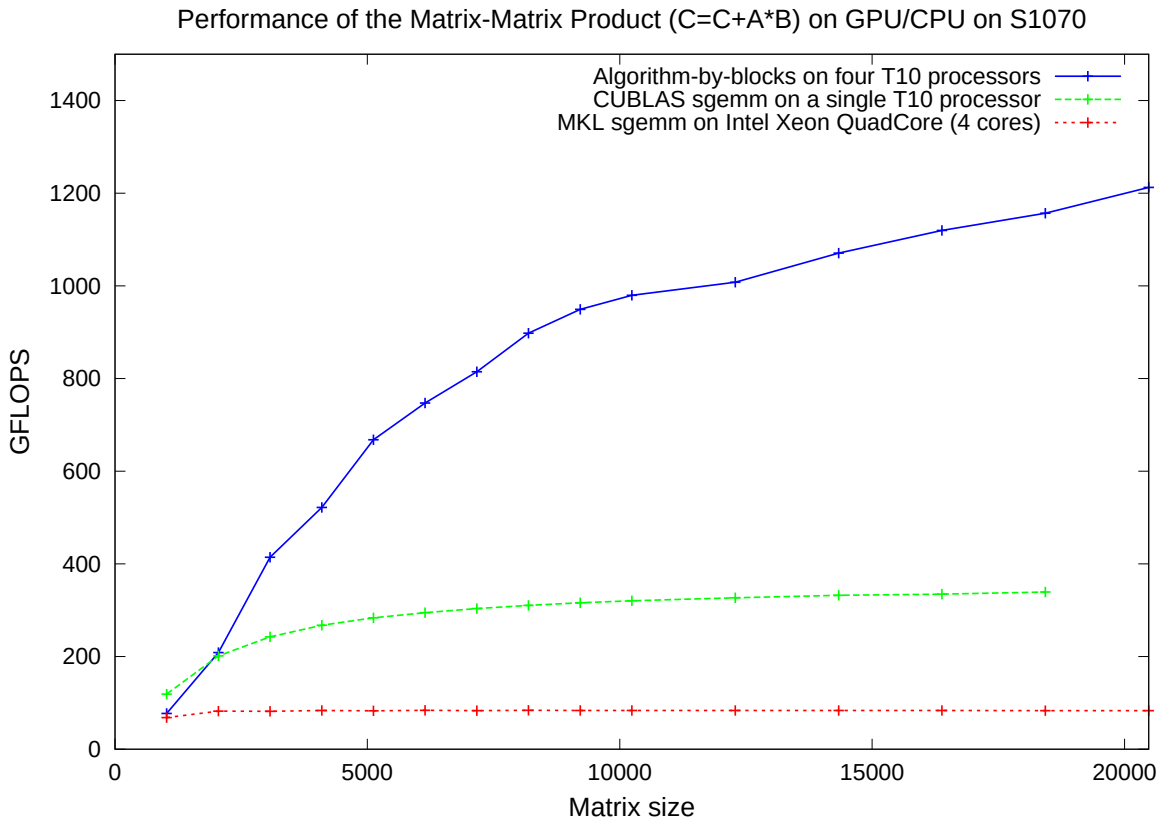
- Efforts to modularize compilers retain the same primtives
  - compiling on the fly (JIT)
  - **L**ow **L**evel **V**irtual **M**achine
- Raise the level of abstraction
  - **F**enics **F**orm **C**ompiler (variational form compiler)
  - Mython (**D**omain **S**pecific **L**anguage generator)

Spiral



- Spiral Team, http://www.spiral.net
- Uses an intermediate language, SPL, and then generates C
- Works by circumscribing the algorithmic domain

FLAME & FLASH



- Robert van de Geijn, http://www.cs.utexas.edu/users/flame
- FLAME is an Algorithm-By-Blocks interface
- FLASH/SuperMatrix is a runtime system

Representation Hierarchy

Divide the work into levels:

- Model
- Algorithm
- Implementation

Representation Hierarchy

Divide the work into levels:

- Model
- Algorithm
- Implementation

Spiral Project:

- **D**iscrete **F**ourier **T**ransform (DSP)
- **F**ast **F**ourier **T**ransform (SPL)
- C Implementation (SPL Compiler)

Representation Hierarchy

Divide the work into levels:

- Model
- Algorithm
- Implementation

FLAME Project:

- Abstract LA (PME/Invariants)
- Basic LA (FLAME/FLASH)
- Scheduling (SuperMatrix)

Representation Hierarchy

Divide the work into levels:

- Model
- Algorithm
- Implementation

FEniCS Project:

- Navier-Stokes (FFC)
- Finite Element (FIAT)
- Integration/Assembly (FErari)

Representation Hierarchy

Divide the work into levels:

- Model
- Algorithm
- Implementation

Treecodes:

- Kernels with decay (Coulomb)
- Treecodes (PetFMM)
- Scheduling (PetFMM-GPU)

Representation Hierarchy

Divide the work into levels:

- Model
- Algorithm
- Implementation

Treecodes:

- Kernels with decay (Coulomb)
- Treecodes (PetFMM)
- Scheduling (PetFMM-GPU)

Each level demands a strong abstraction layer

Outline

Hierarchical Design

Big Idea: **Hierarchy**

Multilevel Method

- Solve local problems
  - Locality of operations is key for efficient implementation
  - Should enable reuse of serial implementation
- Stitch together to form a global solution
  - Manifold or Domain Decomposition idea: local pieces w/ overlap
  - Global complexity is encoded in the (small) Overlap

Hierarchical Design

Big Idea: **Hierarchy**

Multilevel Method

- Solve local problems
  - Locality of operations is key for efficient implementation
  - Should enable reuse of serial implementation
- Stitch together to form a global solution
  - Manifold or Domain Decomposition idea: local pieces w/ overlap
  - Global complexity is encoded in the (small) Overlap

Hierarchical Design

Big Idea: **Hierarchy**

Multilevel Method

- Solve local problems
  - Locality of operations is key for efficient implementation
  - Should enable reuse of serial implementation
- Stitch together to form a global solution
  - Manifold or Domain Decomposition idea: local pieces w/ overlap
  - Global complexity is encoded in the (small) Overlap

Hierarchical Design

Big Idea: **Hierarchy**

Multilevel Method

- Solve local problems
  - Locality of operations is key for efficient implementation
  - Should enable reuse of serial implementation
- Stitch together to form a global solution
  - Manifold or Domain Decomposition idea: local pieces w/ overlap
  - Global complexity is encoded in the (small) Overlap

Hierarchical Design

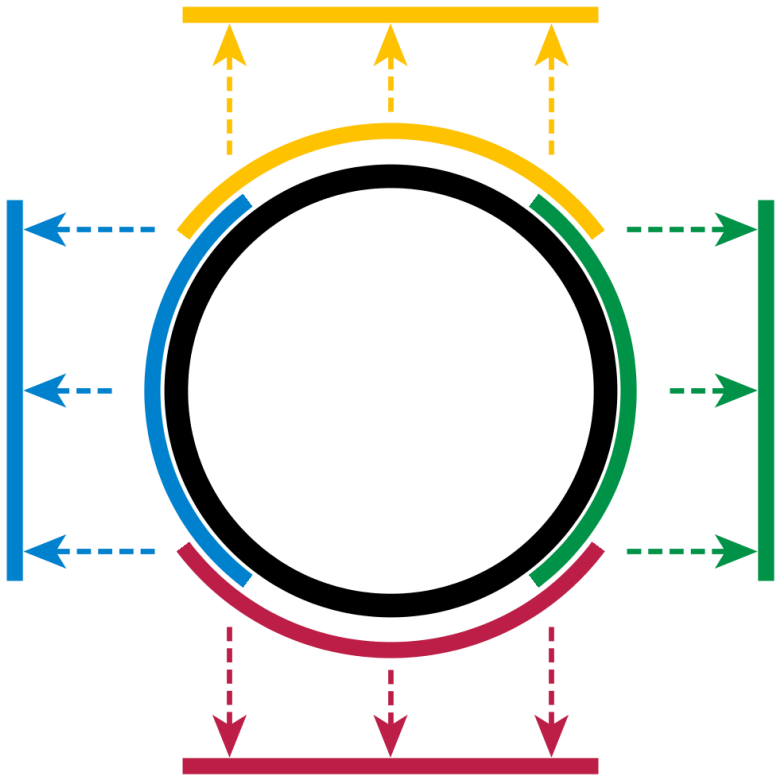
Big Idea: **Hierarchy**

Multilevel Method

- Solve local problems
  - Locality of operations is key for efficient implementation
  - Should enable reuse of serial implementation
- Stitch together to form a global solution
  - Manifold or Domain Decomposition idea: local pieces w/ overlap
  - Global complexity is encoded in the (small) Overlap

Example: Manifold

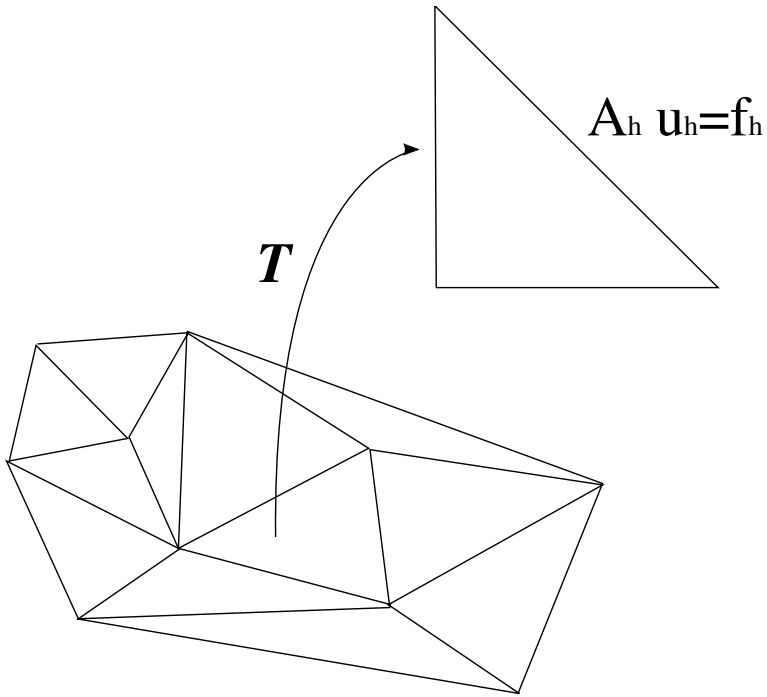
Manifolds are locally homeomorphic to $\mathbb{R}^n$:



Transition maps provide a mechanism to connect the pieces.

Example: FEM

The Finite Element Method does computation in a local basis:



The operator $\mathcal{T}$ maps between the local and global bases.

Global and Local

Local (analytical)

- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
- Largely dim dependent
  (e.g. quadrature)

Global (topological)

- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent
  (e.g. mesh traversal)

Global and Local

Local (analytical)

- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
- Largely dim dependent (e.g. quadrature)

Global (topological)

- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent (e.g. mesh traversal)

Global and Local

Local (analytical)

- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
- Largely dim dependent
  (e.g. quadrature)

Global (topological)

- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent
  (e.g. mesh traversal)

Global and Local

Local (analytical)

- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
- Largely dim dependent
  (e.g. quadrature)

Global (topological)

- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent
  (e.g. mesh traversal)

Global and Local

Local (analytical)

- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
- Largely dim dependent (e.g. quadrature)

Global (topological)

- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent (e.g. mesh traversal)

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Claim: Hierarchical operations can be handled by a single interface

Why Optimal Algorithms?

- The more powerful the computer,
  the greater the importance of optimality
- Example:
  - Suppose Alg_1 solves a problem in time CN^2, N is the input size
  - Suppose Alg_2 solves the same problem in time CN
  - Suppose Alg_1 and Alg_2 are able to use 10,000 processors
- In constant time compared to serial,
  - Alg1 can run a problem 100X larger
  - Alg2 can run a problem 10,000X larger
- Alternatively, filling the machine's memory,
  - Alg1 requires 100X time
  - Alg2 runs in constant time

Sieve Overview

- Hierarchy is the centerpiece
  - Strip out unneeded complexity (dimension, shape, . . .)
- Single relation, covering, handles all hierarchy
  - Rich enough for FEM
- Single operation, completion, for parallelism
  - Enforces consistency of the relation

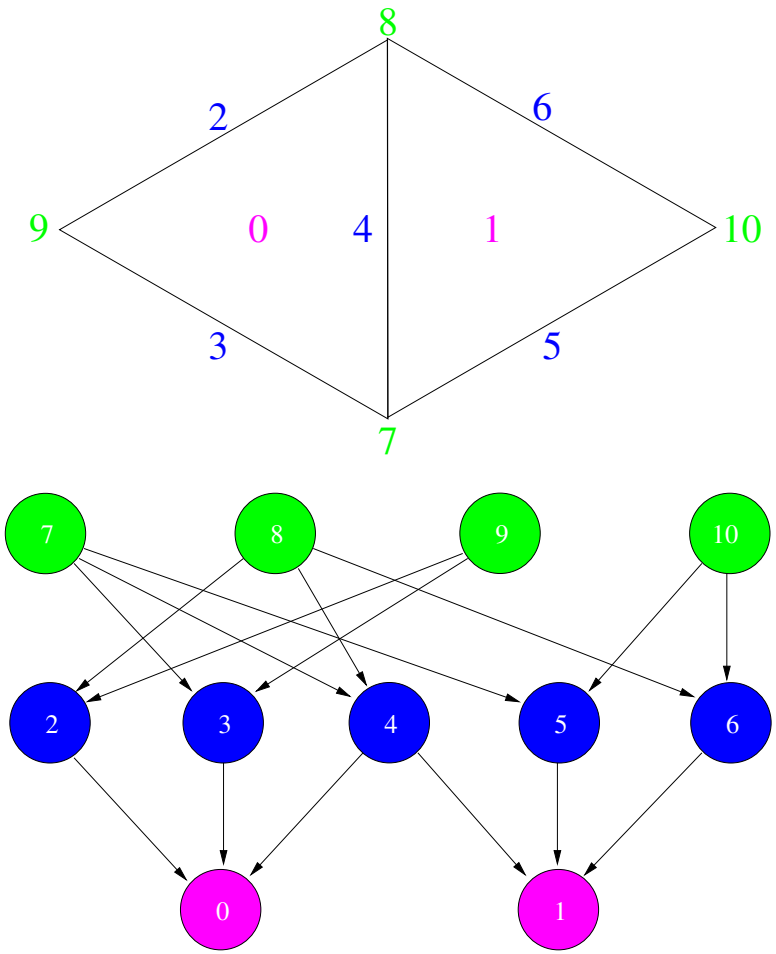
Sieve Overview

- Hierarchy is the centerpiece
  - Strip out unneeded complexity (dimension, shape, . . .)
- Single relation, covering, handles all hierarchy
  - Rich enough for FEM
- Single operation, completion, for parallelism
  - Enforces consistency of the relation

Sieve Overview

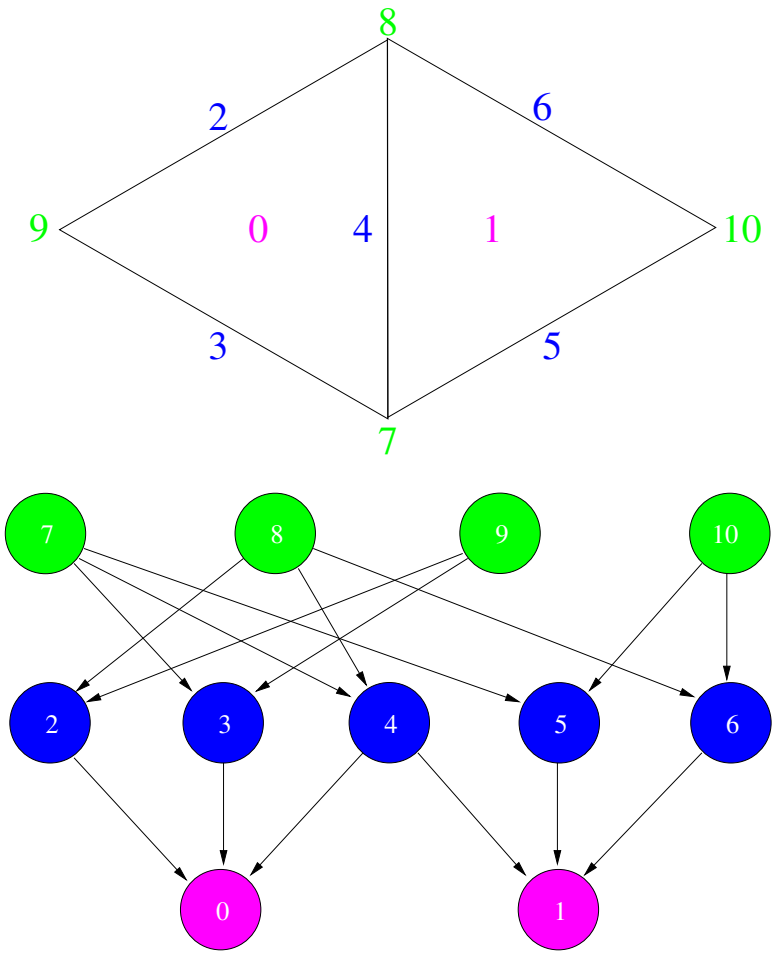
- Hierarchy is the centerpiece
  - Strip out unneeded complexity (dimension, shape, . . .)
- Single relation, covering, handles all hierarchy
  - Rich enough for FEM
- Single operation, completion, for parallelism
  - Enforces consistency of the relation

Doublet Mesh



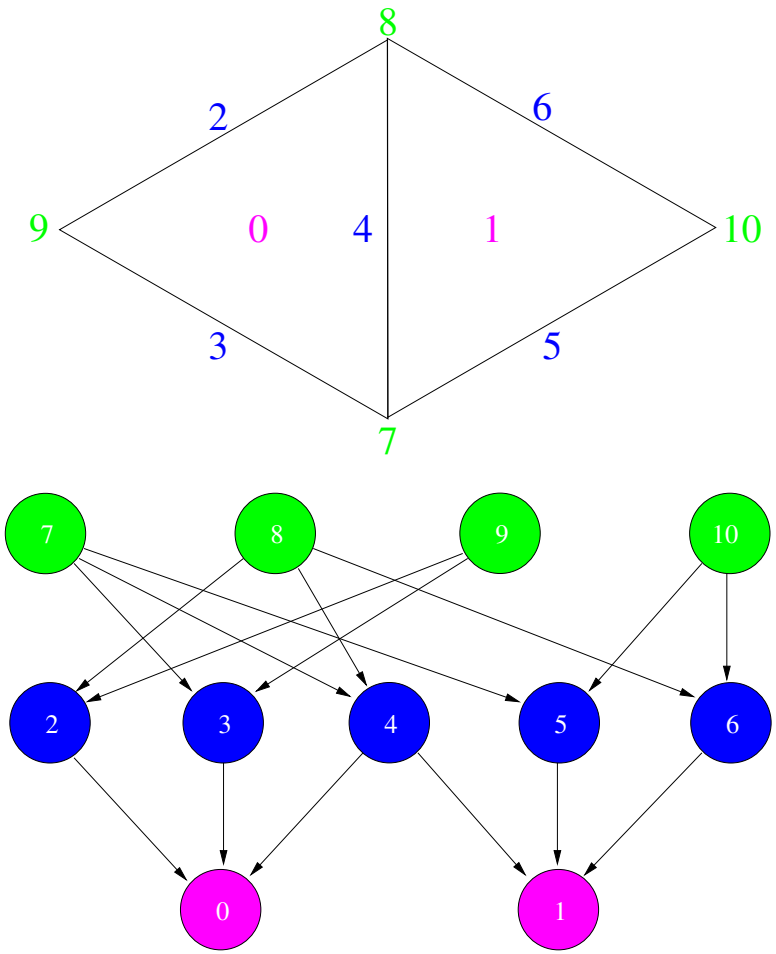
- Incidence/covering arrows
- $cone(0) = \{2, 3, 4\}$
- $support(7) = \{2, 3\}$

Doublet Mesh



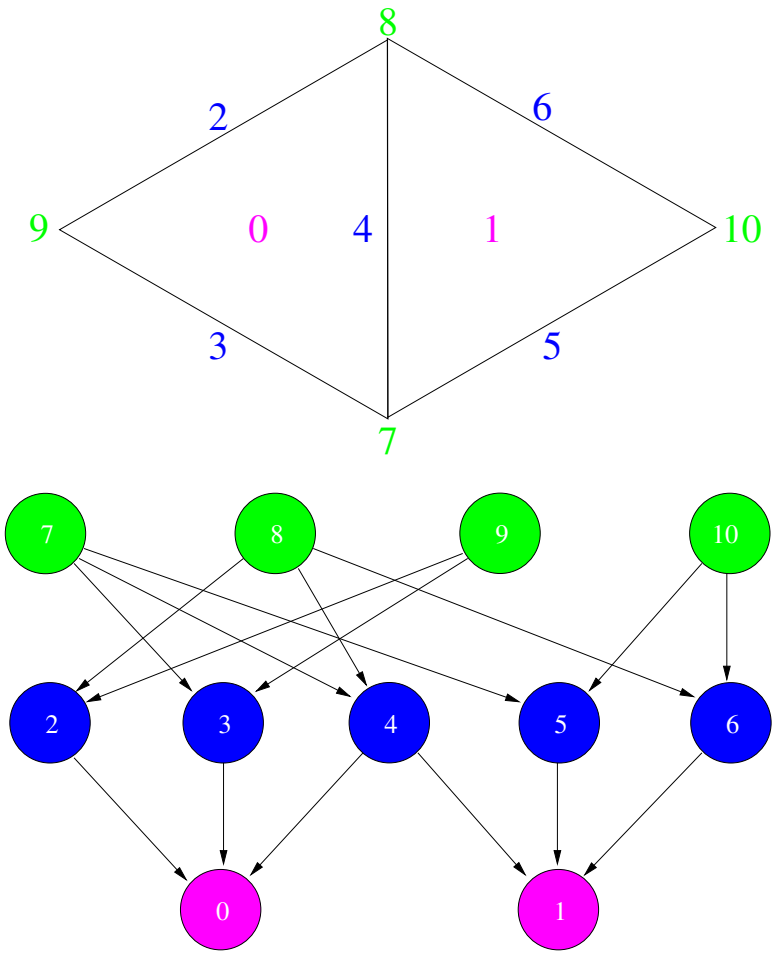
- Incidence/covering arrows
- $cone(0) = \{2, 3, 4\}$
- $support(7) = \{2, 3\}$

Doublet Mesh



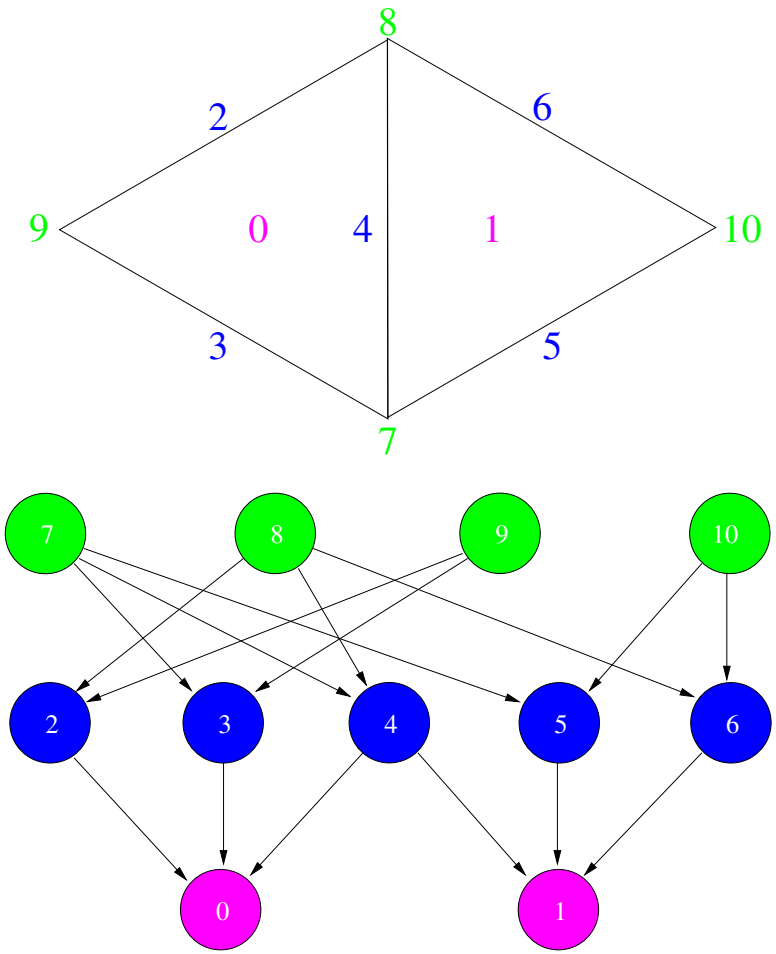
- Incidence/covering arrows
- $cone(0) = \{2, 3, 4\}$
- $support(7) = \{2, 3\}$

Doublet Mesh



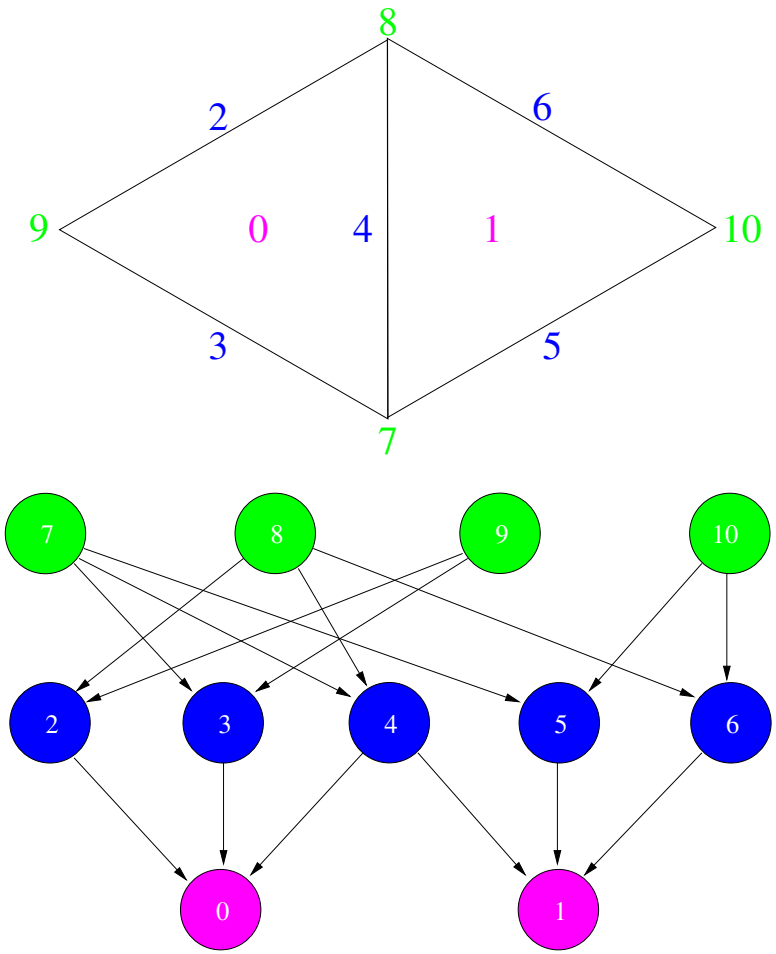
- Incidence/covering arrows
- $closure(0) = \{0, 2, 3, 4, 7, 8, 9\}$
- $star(7) = \{7, 2, 3, 0\}$

Doublet Mesh



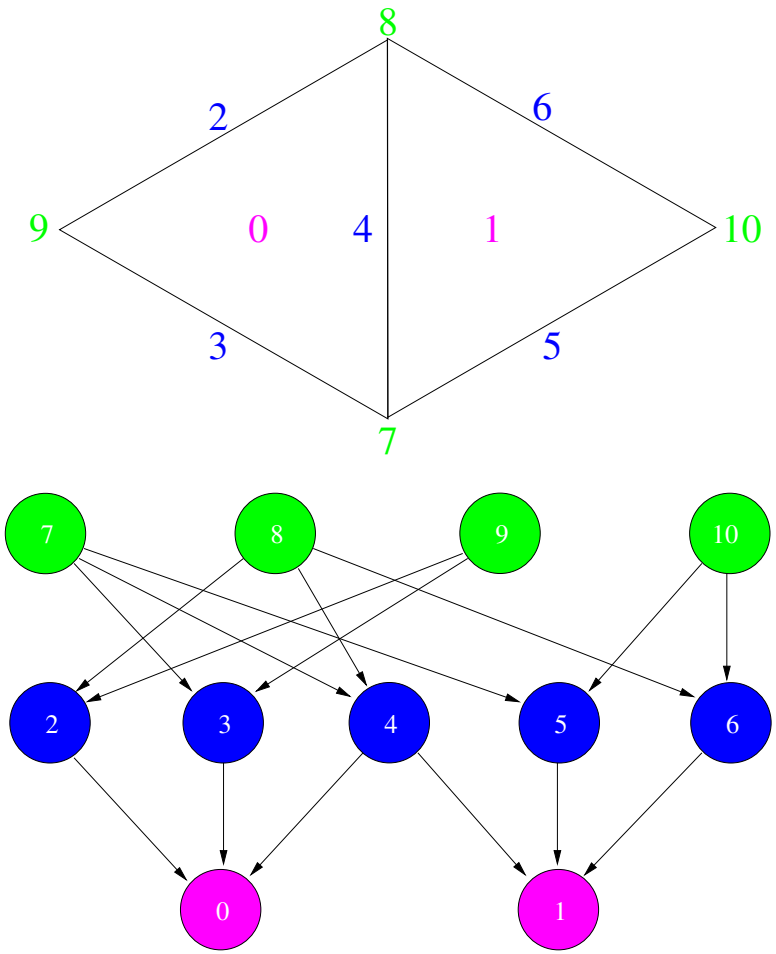
- Incidence/covering arrows
- $closure(0) = \{0, 2, 3, 4, 7, 8, 9\}$
- $star(7) = \{7, 2, 3, 0\}$

Doublet Mesh



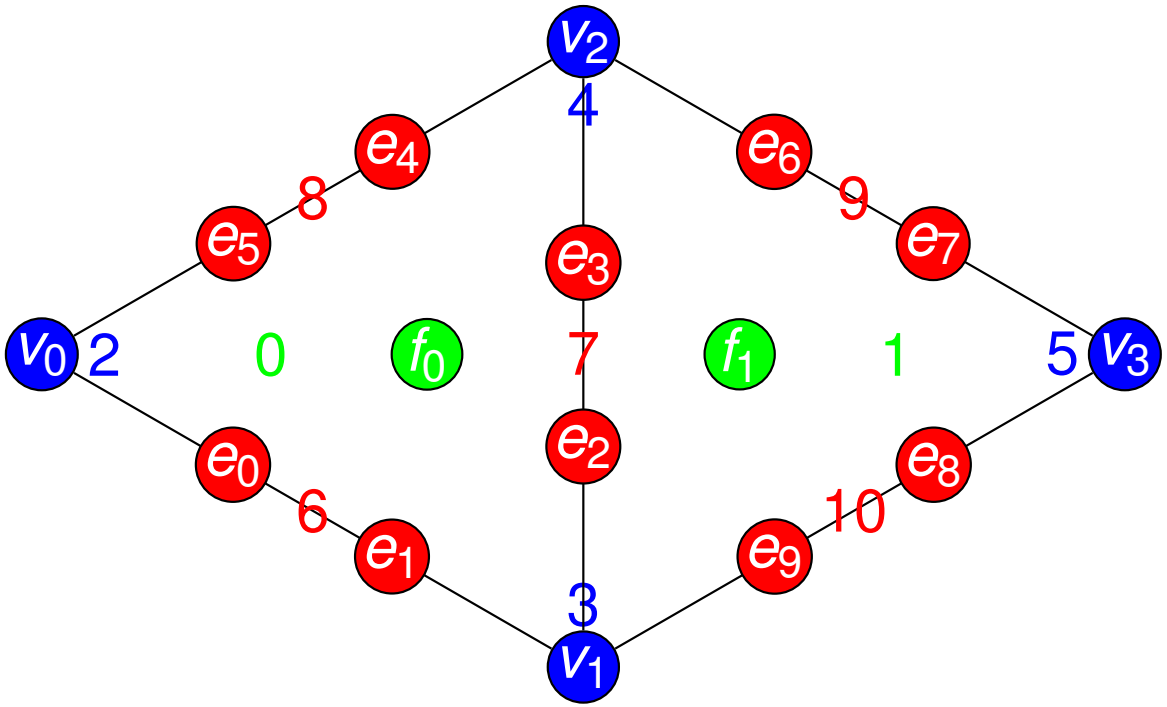
- Incidence/covering arrows
- $meet(0, 1) = \{4\}$
- $join(8, 9) = \{4\}$

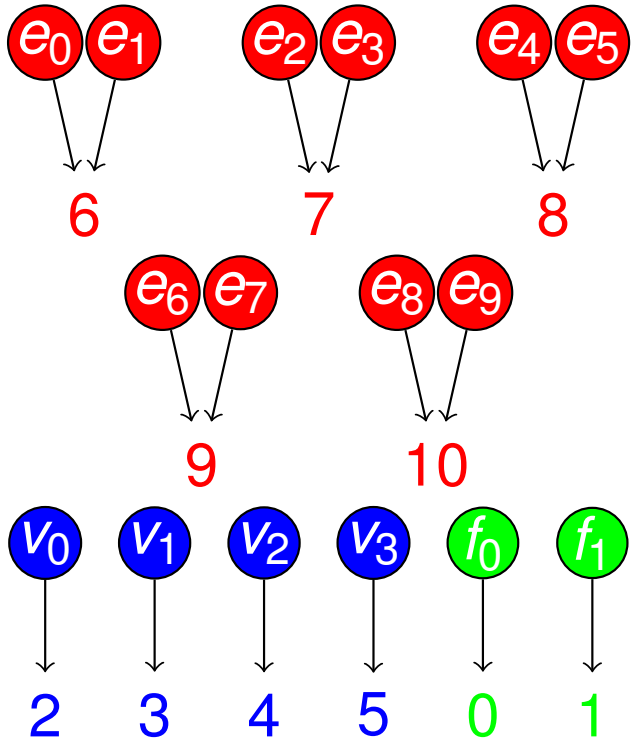
Doublet Mesh



- Incidence/covering arrows
- $meet(0, 1) = \{4\}$
- $join(8, 9) = \{4\}$

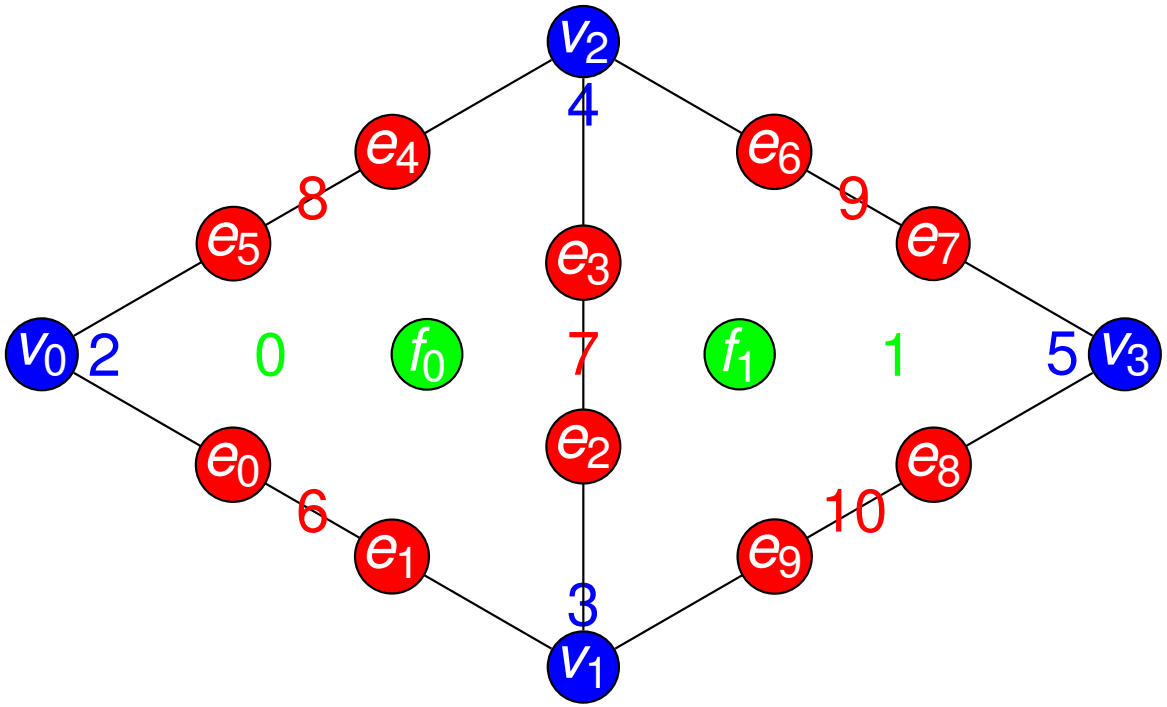
Doublet Section

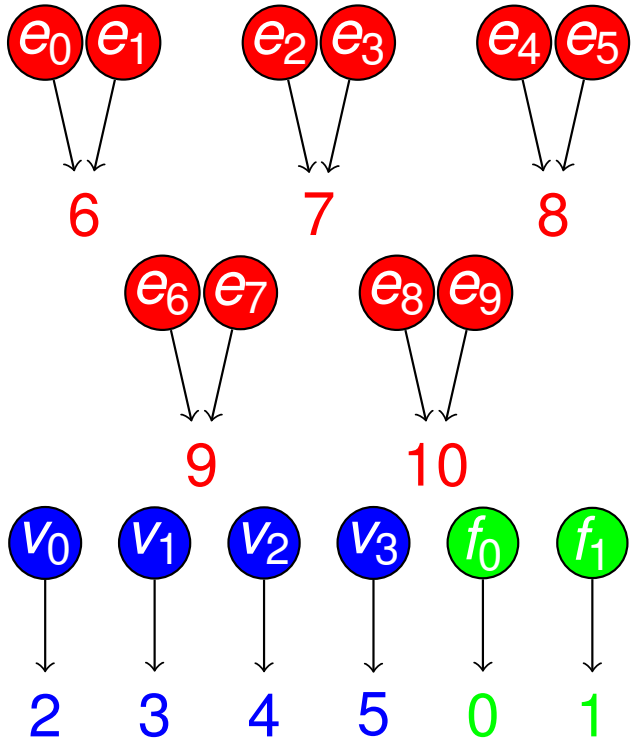




- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

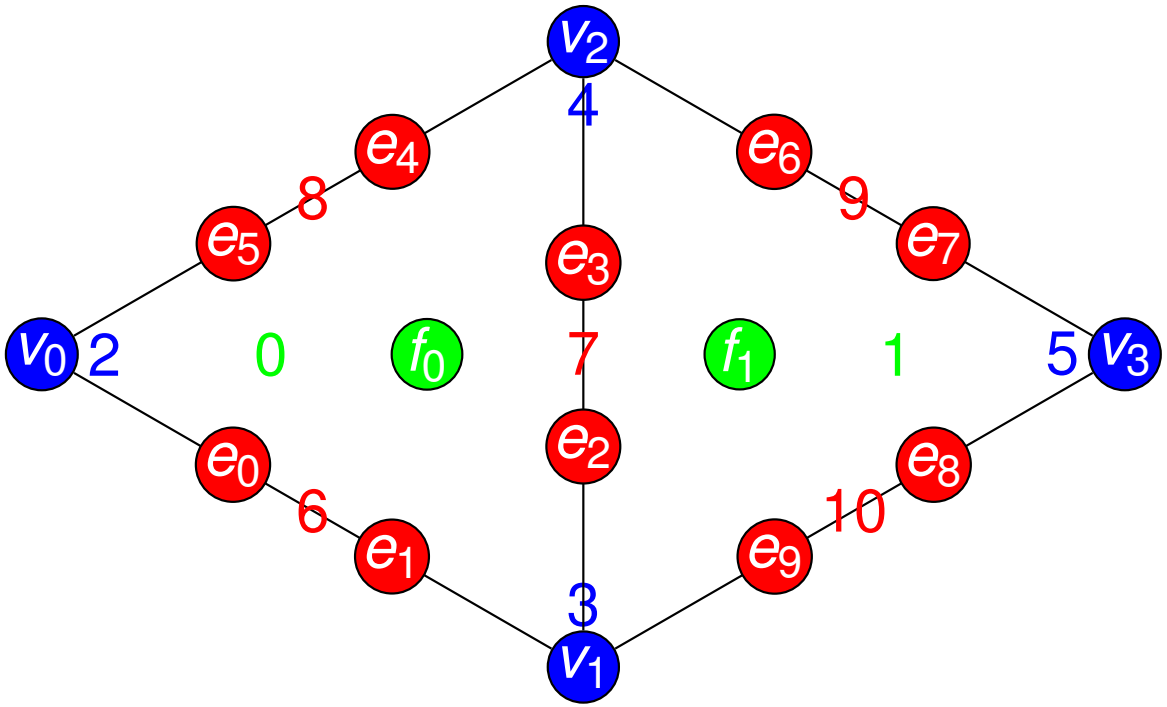
Doublet Section

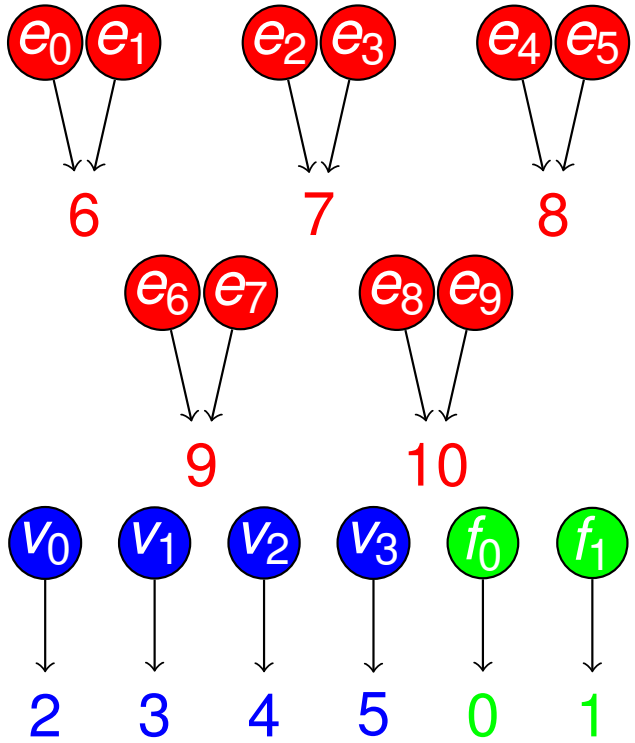




- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

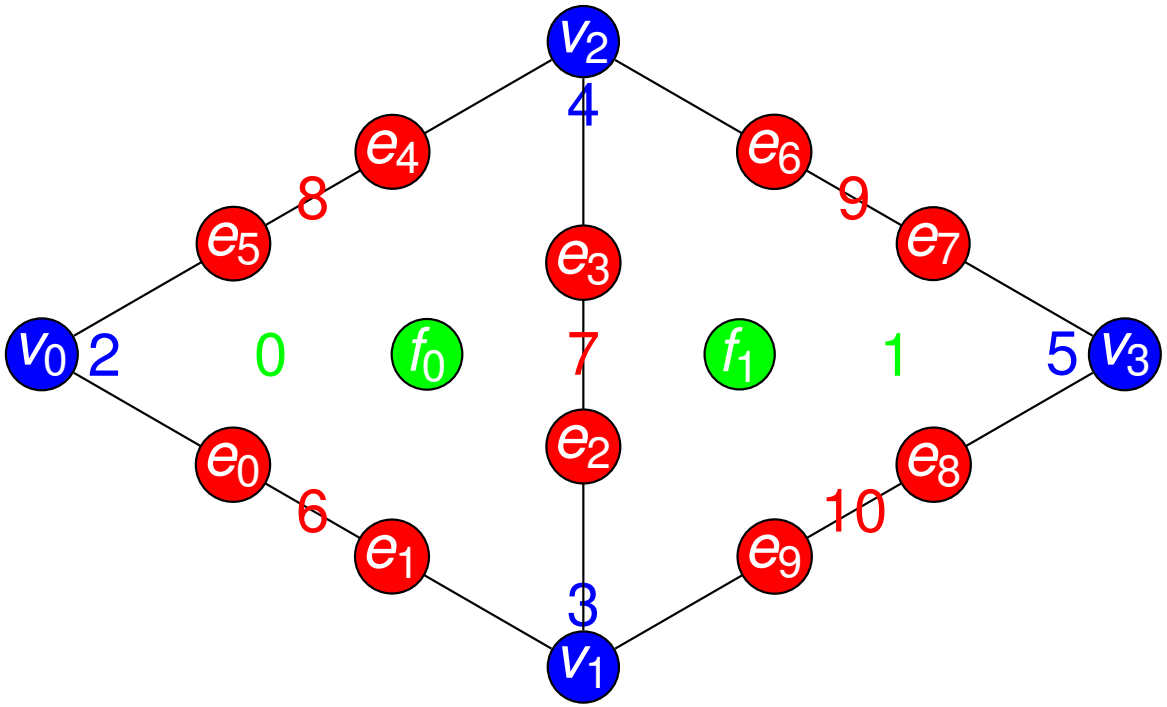
Doublet Section

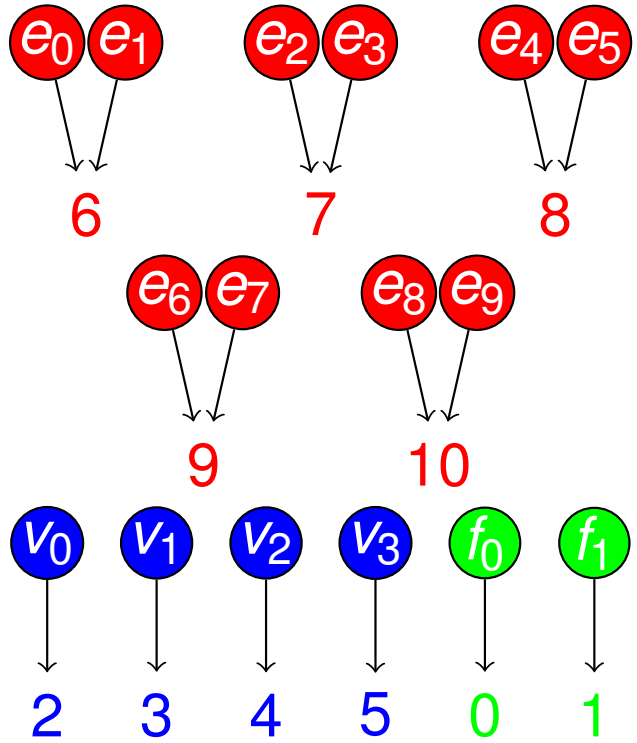




- **Section** interface
  - *restrict*(0) = $\{f_0\}$
  - *restrict*(2) = $\{v_0\}$
  - *restrict*(6) = $\{e_0, e_1\}$

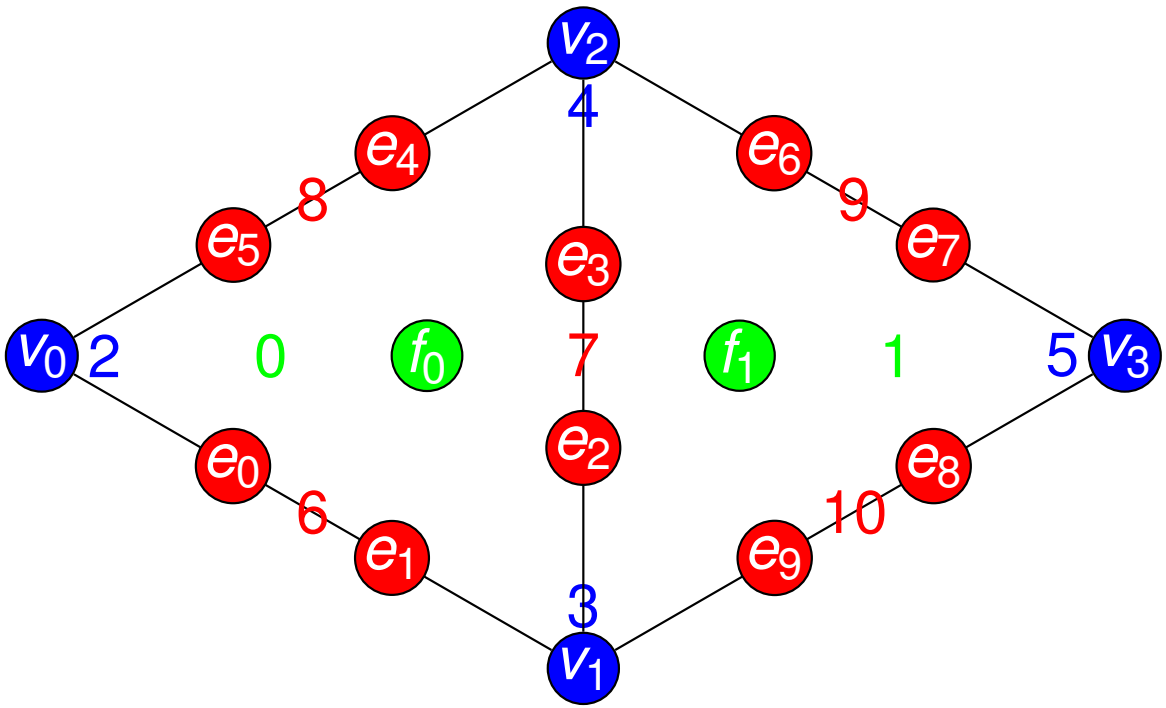
Doublet Section

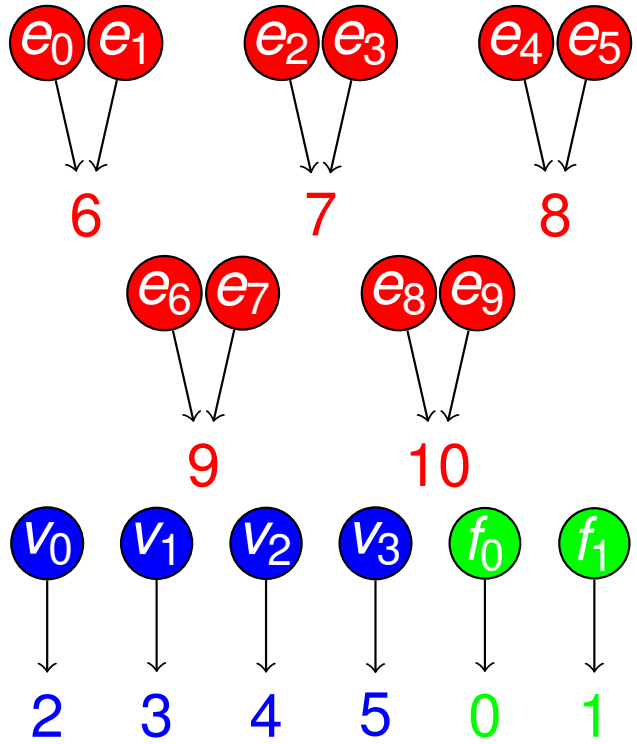




- **Section** interface
  - *restrict*(0) = $\{f_0\}$
  - *restrict*(2) = $\{v_0\}$
  - *restrict*(6) = $\{e_0, e_1\}$

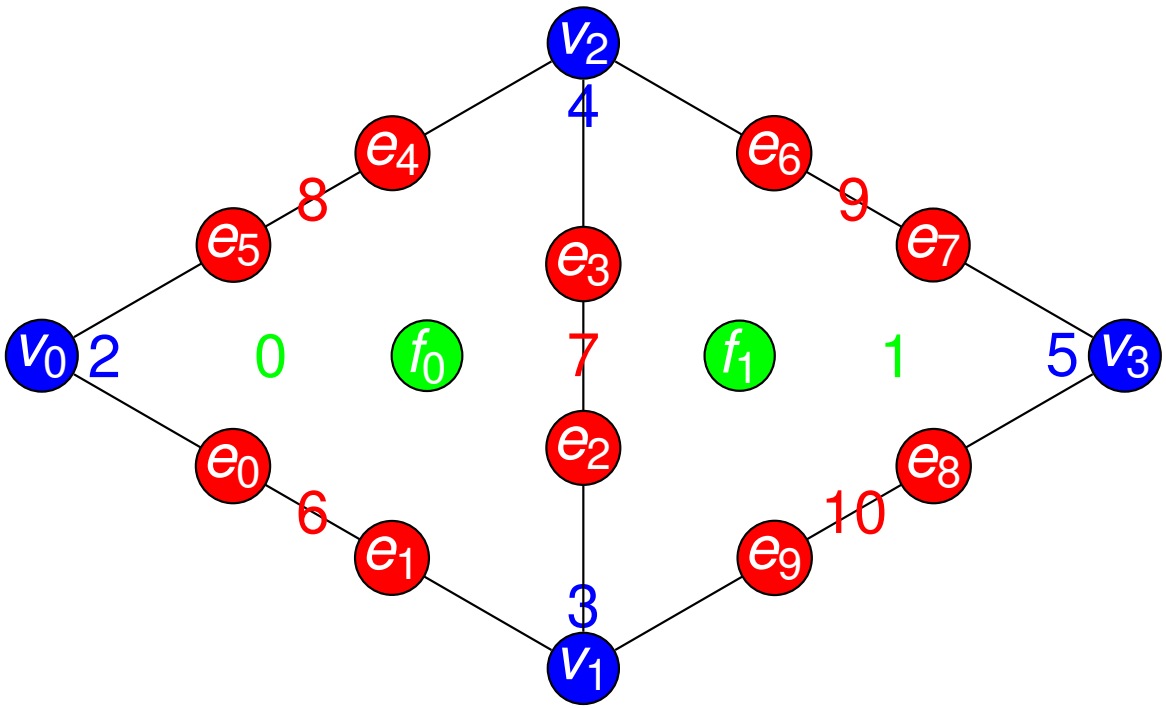
Doublet Section

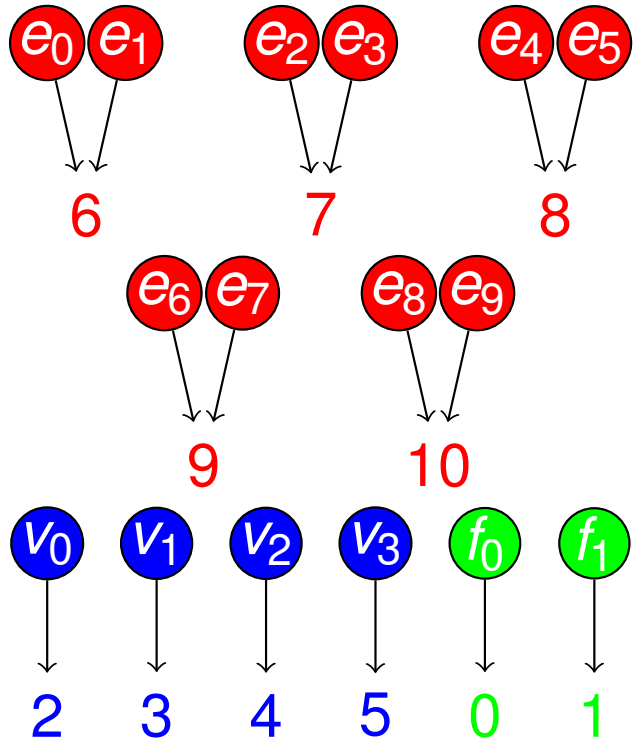




- Topological traversals: follow connectivity
  - $restrictClosure(0) = \{f_0\, e_0\, e_1\, e_2\, e_3\, e_4\, e_5\, v_0\, v_1\, v_2\}$
  - $restrictStar(7) = \{v_0\, e_0\, e_1\, e_4\, e_5\, f_0\}$

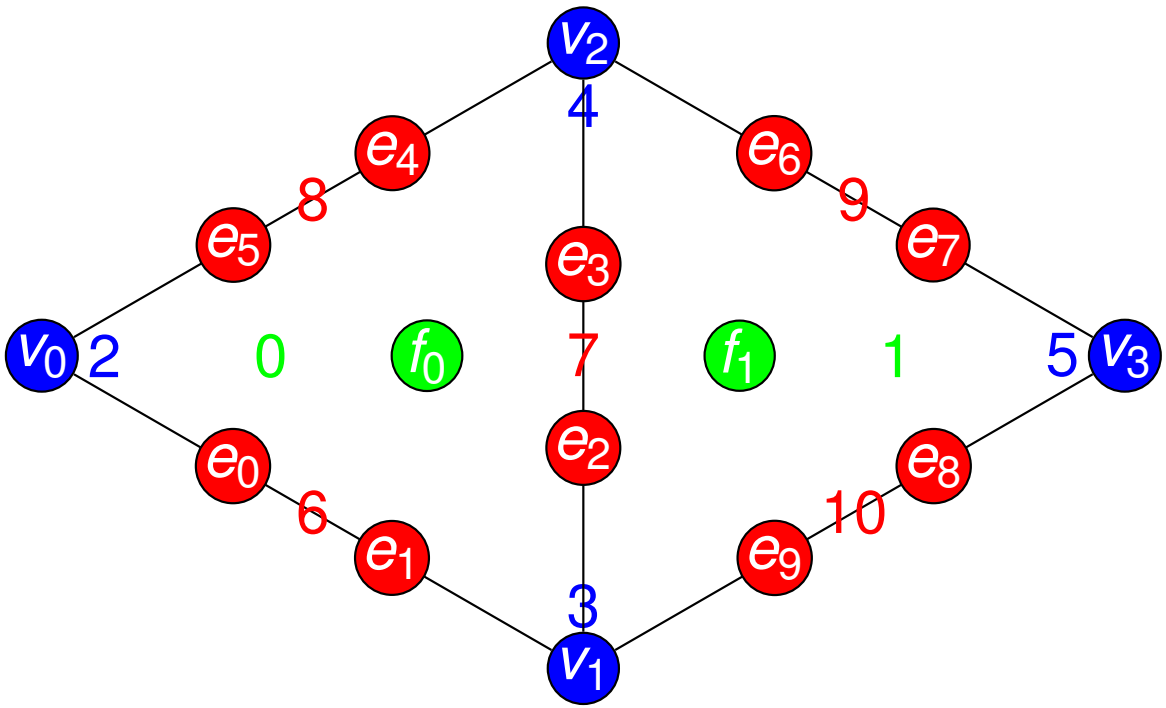
Doublet Section

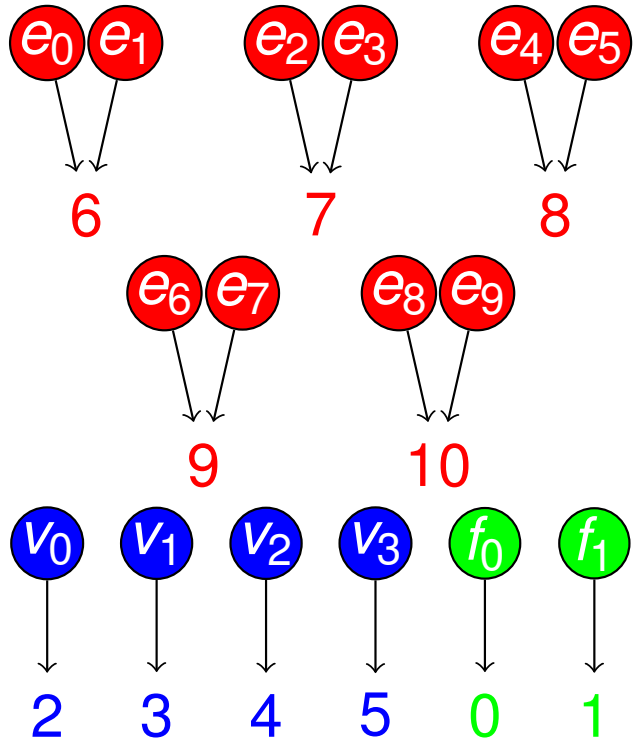




- Topological traversals: follow connectivity
  - $\mathit{restrictClosure}(0) = \{f_0\, e_0\, e_1\, e_2\, e_3\, e_4\, e_5\, v_0\, v_1\, v_2\}$
  - $\mathit{restrictStar}(7) = \{v_0\, e_0\, e_1\, e_4\, e_5\, f_0\}$

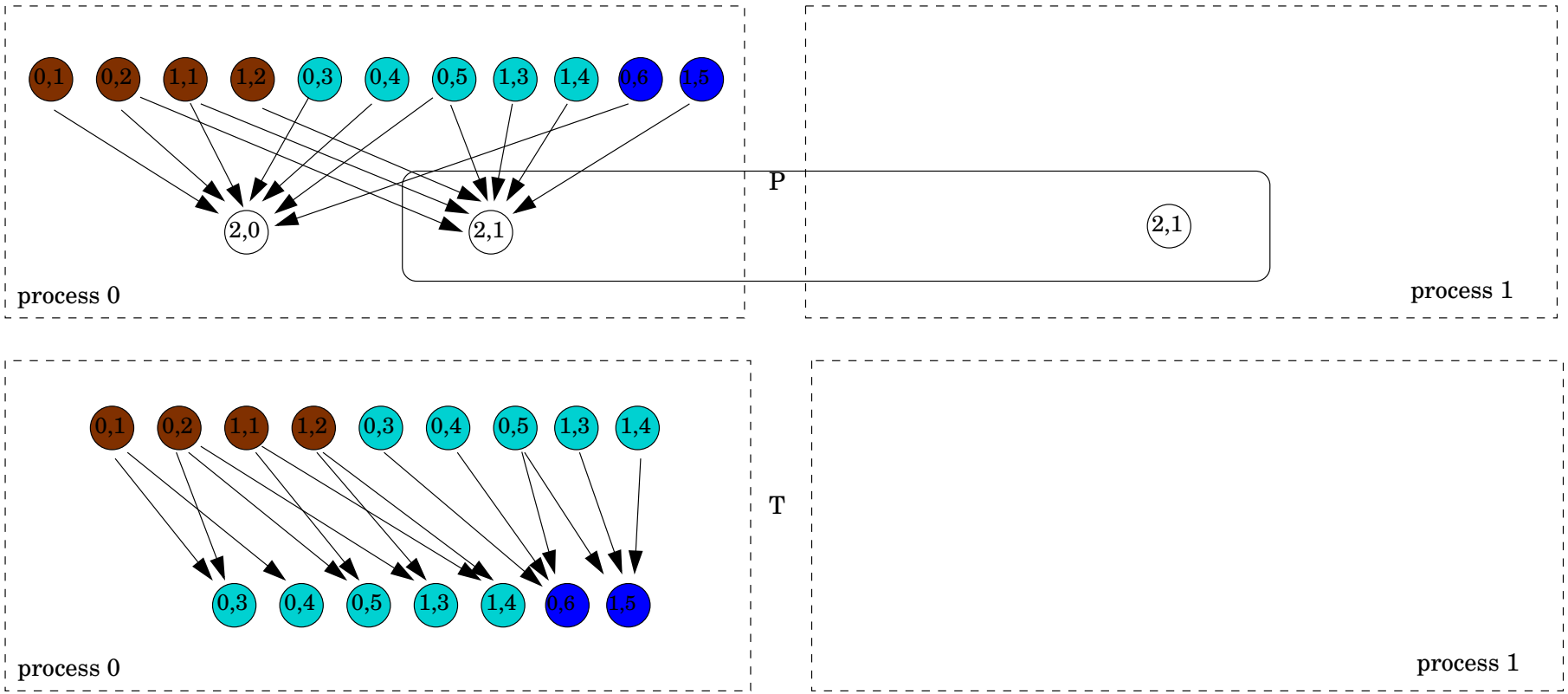
Doublet Section



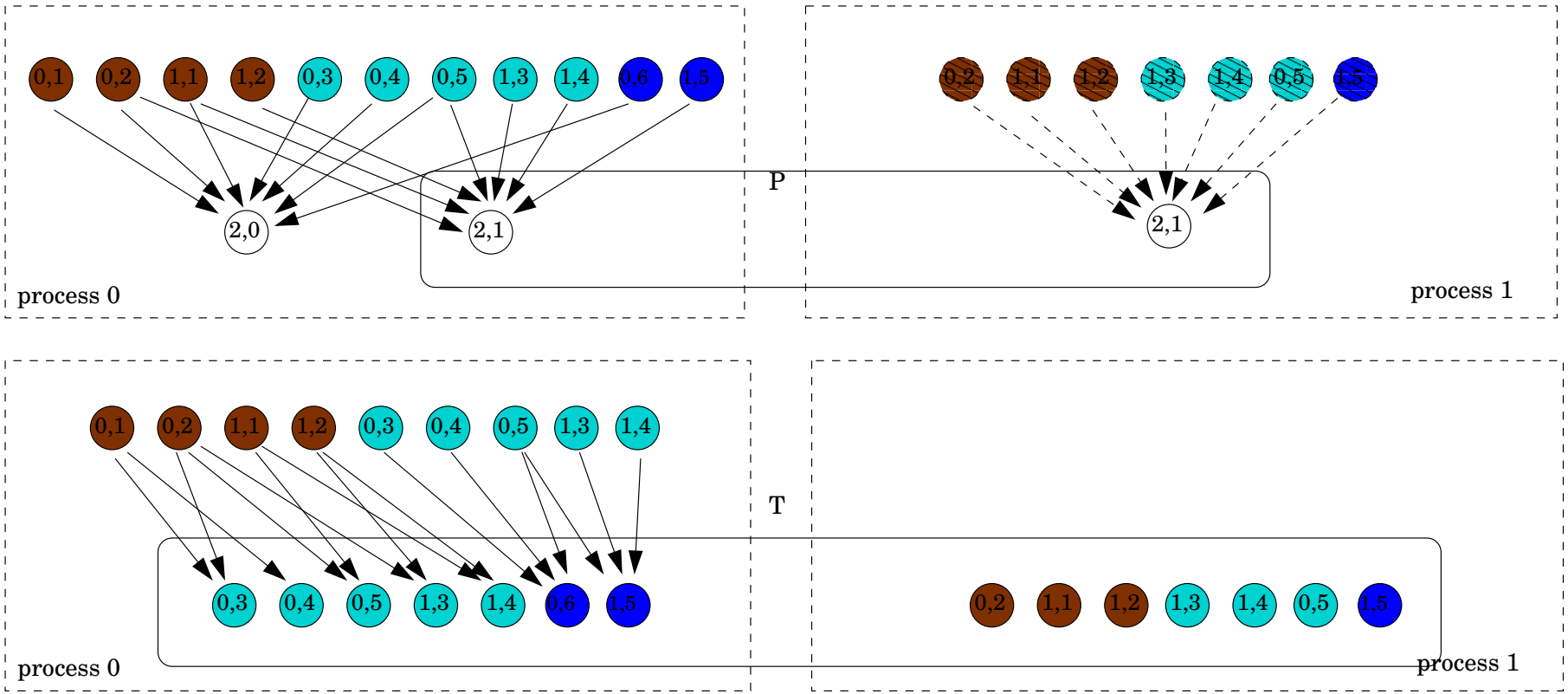


- Topological traversals: follow connectivity
  - $\mathit{restrictClosure}(0) = \{f_0\, e_0\, e_1\, e_2\, e_3\, e_4\, e_5\, v_0\, v_1\, v_2\}$
  - $\mathit{restrictStar}(7) = \{v_0\, e_0\, e_1\, e_4\, e_5\, f_0\}$

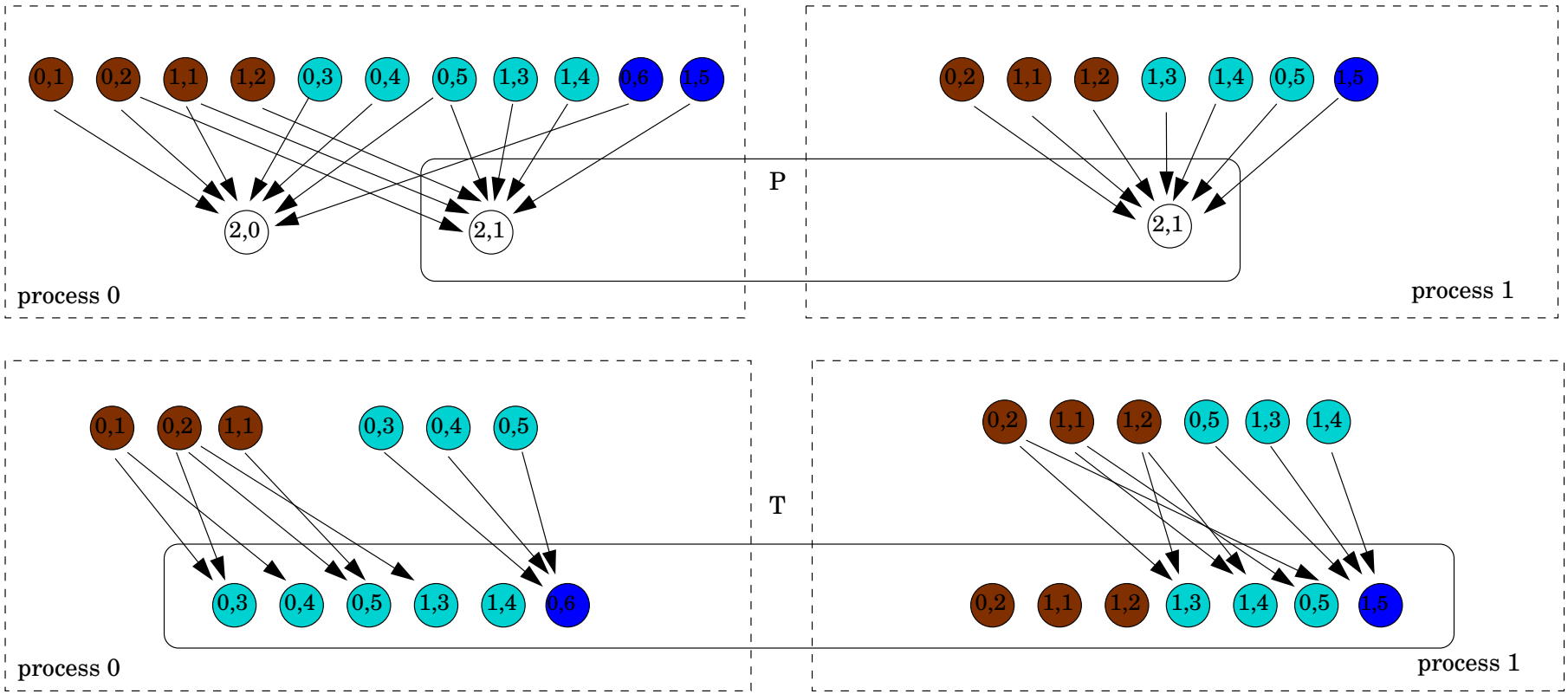
Doublet Mesh Distribution



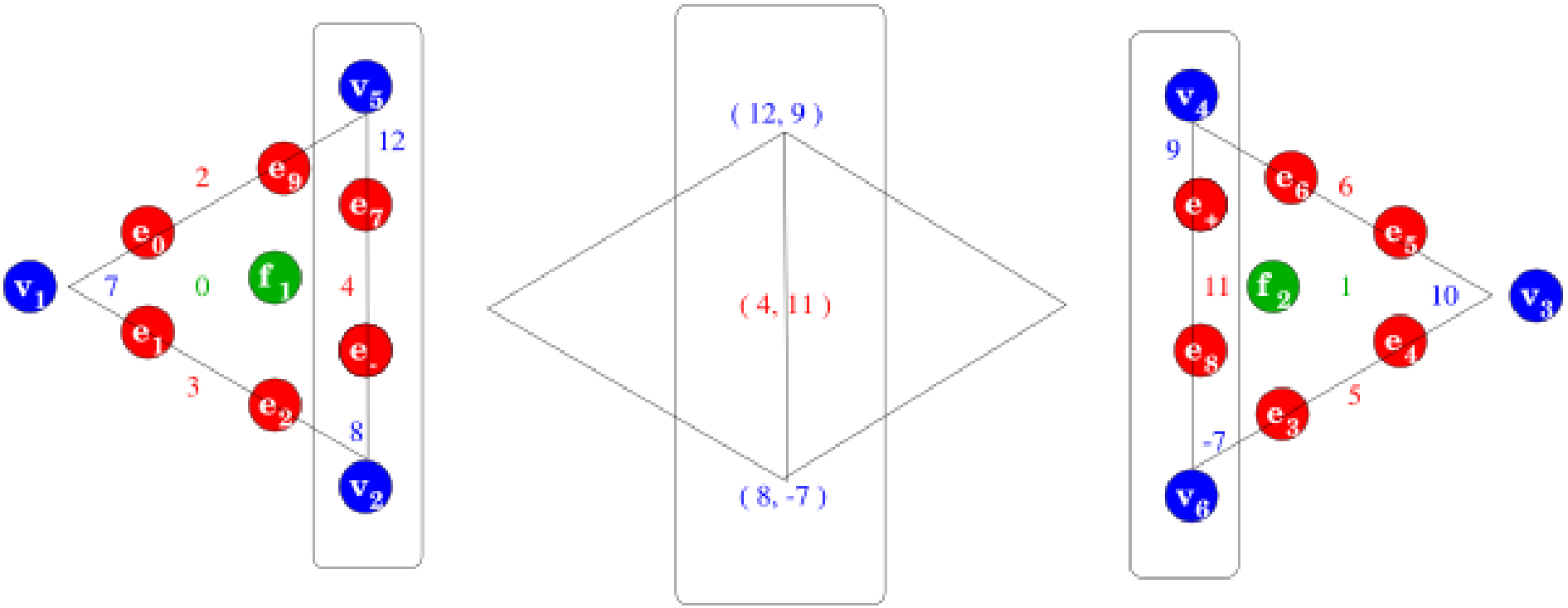
Doublet Mesh Distribution



Doublet Mesh Distribution

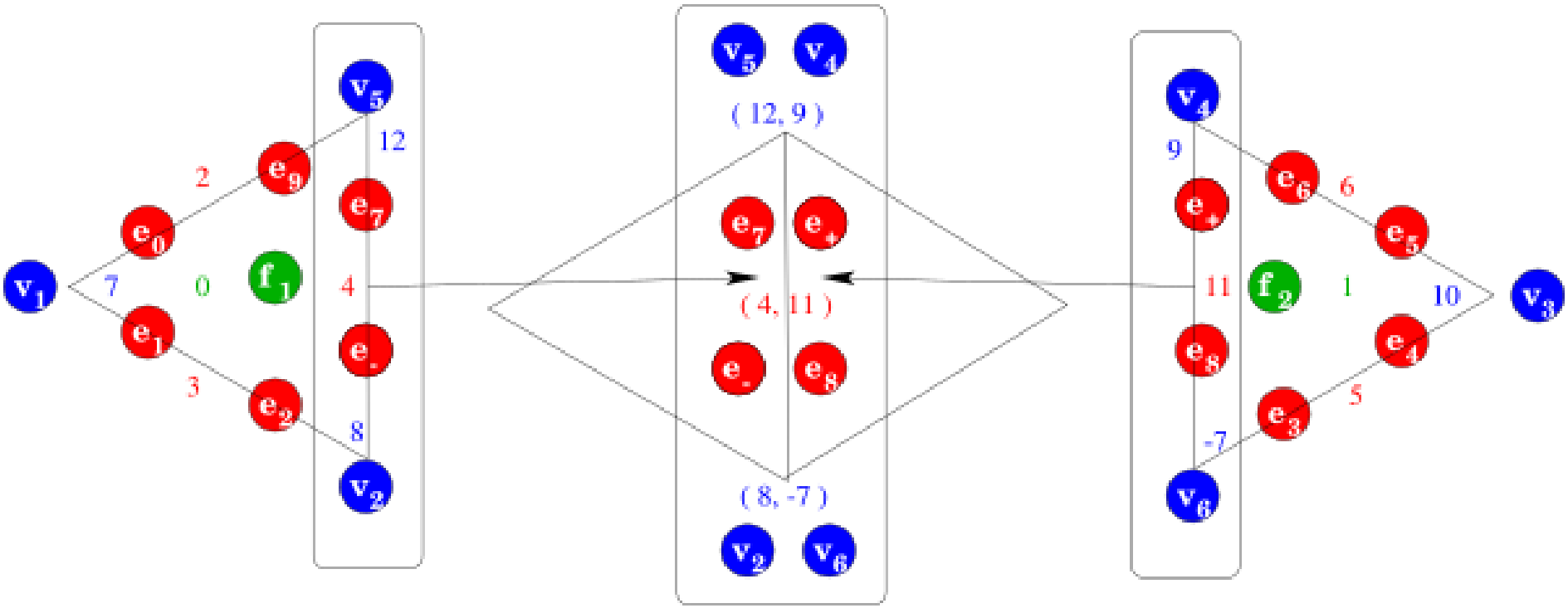


Restriction



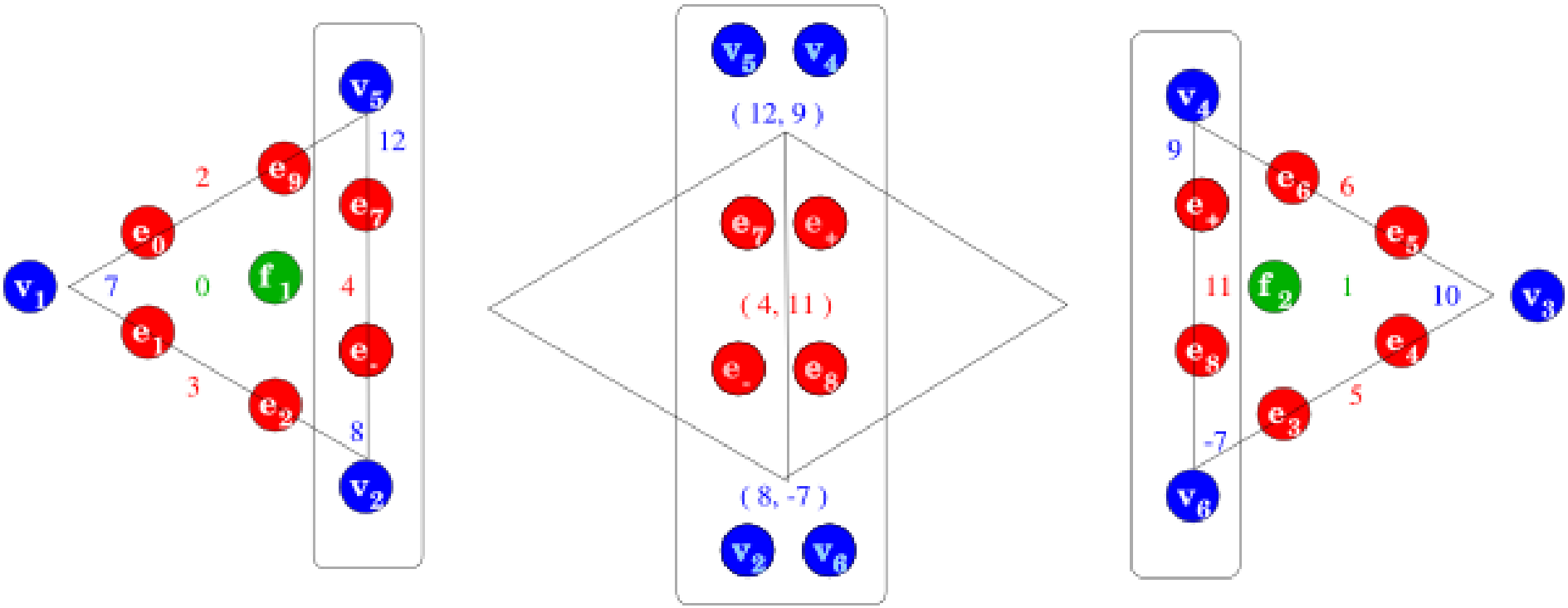
- Localization
  - Restrict to patches (here an edge closure)
  - Compute locally

Delta



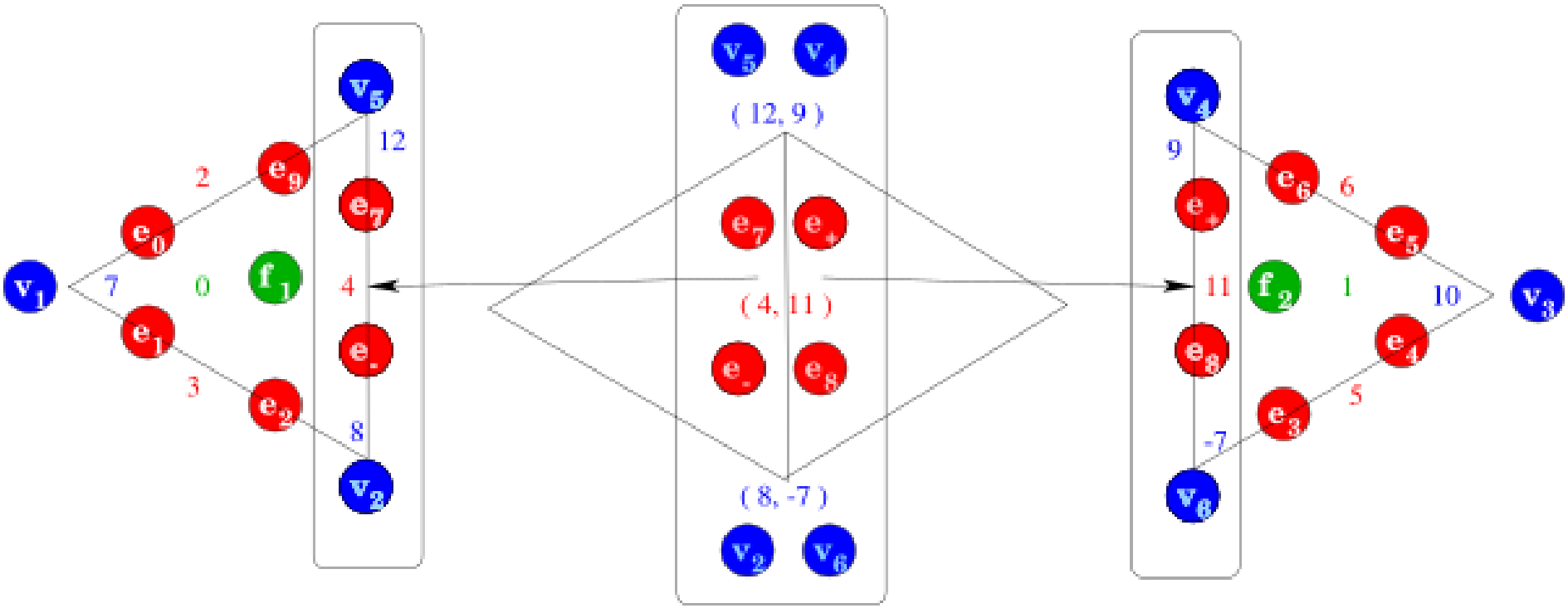
- `Delta`
  - Restrict further to the overlap
  - `Overlap` now carries twice the data

Fusion



- Merge/reconcile data on the overlap
  - Addition (FEM)
  - Replacement (FD)
  - Coordinate transform (Sphere)
  - Linear transform (MG)

Update



- `Update`
  - Update local patch data
  - `Completion` = restrict $\longrightarrow$ fuse $\longrightarrow$ update, in parallel

Uses

Completion has many uses:

FEM accumulating integrals on shared faces

FVM accumulating fluxes on shared cells

FDM setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumlating matvec for a partially assembled matrix

Uses

Completion has many uses:

FEM accumulating integrals on shared faces

FVM accumulating fluxes on shared cells

FDM setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumlating matvec for a partially assembled matrix

Uses

Completion has many uses:

FEM accumulating integrals on shared faces

FVM accumulating fluxes on shared cells

FDM setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumlating matvec for a partially assembled matrix

Uses

Completion has many uses:

FEM accumulating integrals on shared faces

FVM accumulating fluxes on shared cells

FDM setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumlating matvec for a partially assembled matrix

Uses

Completion has many uses:

FEM accumulating integrals on shared faces

FVM accumulating fluxes on shared cells

FDM setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumlating matvec for a partially assembled matrix

Uses

Completion has many uses:

FEM accumulating integrals on shared faces

FVM accumulating fluxes on shared cells

FDM setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumlating matvec for a partially assembled matrix

Uses

Completion has many uses:

FEM accumulating integrals on shared faces

FVM accumulating fluxes on shared cells

FDM setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumlating matvec for a partially assembled matrix

Benefits

Better mathematical abstractions bring concrete benefits

- Vast reduction in complexity
  - Declarative, rather than imperative, specification
  - Dimension independent code
- Opportunites for optimization
  - Higher level operations missed by traditional compilers
  - Single communication routine to optimize
- Expansion of capabilities
  - Easy model definition
  - Arbitrary elements
  - Complex geometries and embedded boundaries

Part II

Global Computation: Theory

Outline

Hierarchical Design

Big Idea: **Hierarchy**

Multilevel Method

- Solve local problems
  - Locality of operations is key for efficient implementation
  - Should enable reuse of serial implementation
- Stitch together to form a global solution
  - Manifold or Domain Decomposition idea: local pieces w/ overlap
  - Global complexity is encoded in the (small) Overlap

Hierarchical Design

Big Idea: **Hierarchy**

Multilevel Method

- Solve local problems
  - Locality of operations is key for efficient implementation
  - Should enable reuse of serial implementation
- Stitch together to form a global solution
  - Manifold or Domain Decomposition idea: local pieces w/ overlap
  - Global complexity is encoded in the (small) Overlap

Hierarchical Design

Big Idea: **Hierarchy**

Multilevel Method

- Solve local problems
  - Locality of operations is key for efficient implementation
  - Should enable reuse of serial implementation
- Stitch together to form a global solution
  - Manifold or Domain Decomposition idea: local pieces w/ overlap
  - Global complexity is encoded in the (small) Overlap

Hierarchical Design

Big Idea: **Hierarchy**

Multilevel Method

- Solve local problems
  - Locality of operations is key for efficient implementation
  - Should enable reuse of serial implementation
- Stitch together to form a global solution
  - Manifold or Domain Decomposition idea: local pieces w/ overlap
  - Global complexity is encoded in the (small) Overlap

Hierarchical Design

Big Idea: **Hierarchy**

Multilevel Method

- Solve local problems
  - Locality of operations is key for efficient implementation
  - Should enable reuse of serial implementation
- Stitch together to form a global solution
  - Manifold or Domain Decomposition idea: local pieces w/ overlap
  - Global complexity is encoded in the (small) Overlap

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Claim: Hierarchical operations can be handled by a single interface

What Is Optimal?

I will define *optimal* as an $\mathcal{O}(N)$ solution algorithm

These are generally hierarchical, so we need

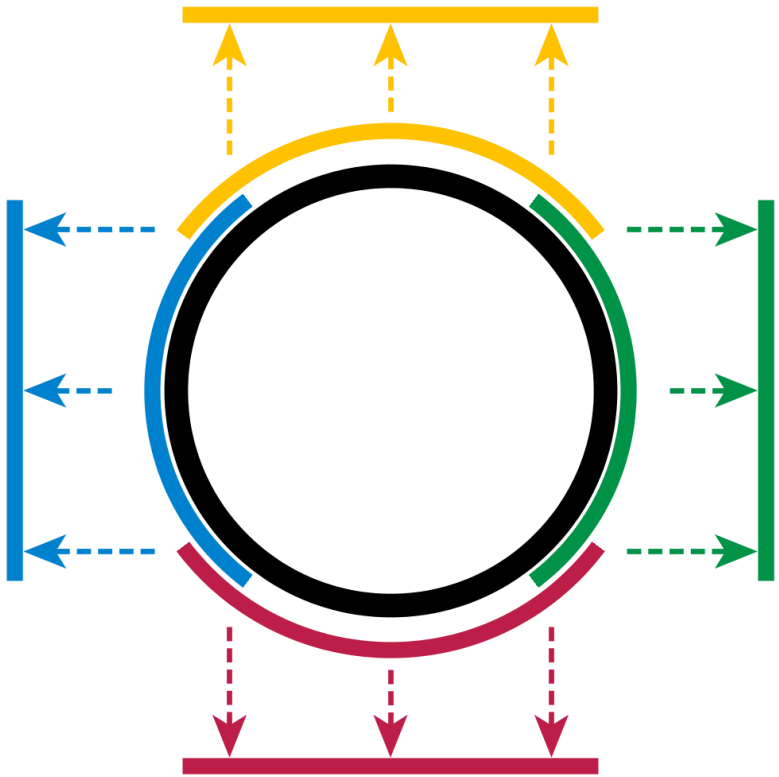
- hierarchy generation
- assembly on subdomains
- restriction and prolongation

Why Optimal Algorithms?

- The more powerful the computer,
  the greater the importance of optimality
- Example:
  - Suppose Alg_1 solves a problem in time CN^2, N is the input size
  - Suppose Alg_2 solves the same problem in time CN
  - Suppose Alg_1 and Alg_2 are able to use 10,000 processors
- In constant time compared to serial,
  - Alg1 can run a problem 100X larger
  - Alg2 can run a problem 10,000X larger
- Alternatively, filling the machine's memory,
  - Alg1 requires 100X time
  - Alg2 runs in constant time

Example: Manifold

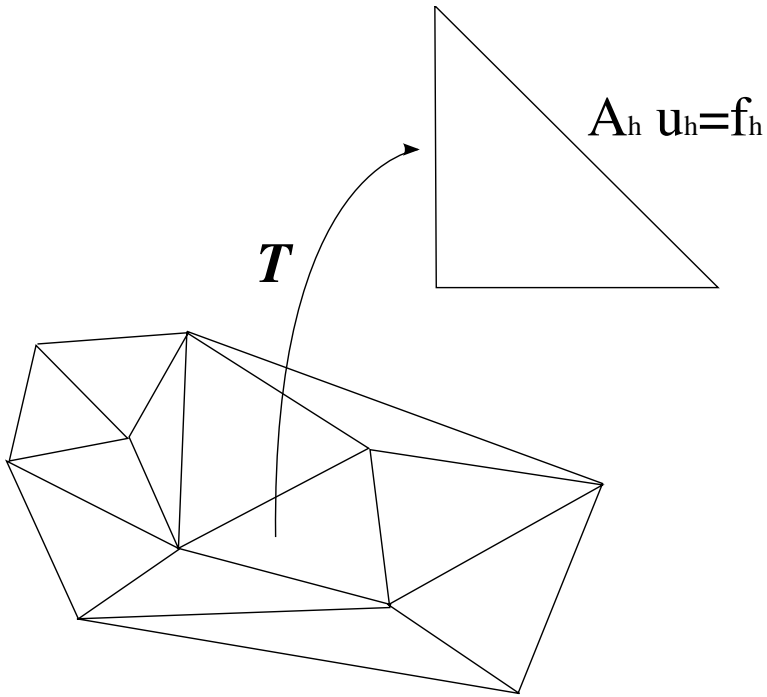
Manifolds are locally homeomorphic to $\mathbb{R}^n$:



Transition maps provide a mechanism to connect the pieces.

Example: FEM

The Finite Element Method does computation in a local basis:



The operator $\mathcal{T}$ maps between the local and global bases.

Outline

Sieve Overview

- Hierarchy is the centerpiece
  - Strip out unneeded complexity (dimension, shape, . . .)
- Single relation, covering, handles all hierarchy
  - Rich enough for FEM
- Single operation, completion, for parallelism
  - Enforces consistency of the relation

Sieve Overview

- Hierarchy is the centerpiece
  - Strip out unneeded complexity (dimension, shape, . . .)
- Single relation, covering, handles all hierarchy
  - Rich enough for FEM
- Single operation, completion, for parallelism
  - Enforces consistency of the relation

Sieve Overview

- Hierarchy is the centerpiece
  - Strip out unneeded complexity (dimension, shape, . . .)
- Single relation, covering, handles all hierarchy
  - Rich enough for FEM
- Single operation, completion, for parallelism
  - Enforces consistency of the relation

Basic Operations

We begin with a basic covering operation:

Basic Operations

We begin with a basic covering operation: `cone()`

Basic Operations

We begin with a basic covering operation: `cone()`

and then add its dual:

Basic Operations

We begin with a basic covering operation: `cone()`

and then add its dual: `support()`

Basic Operations

We begin with a basic covering operation: `cone()`
and then add its dual: `support()`
followed by the transitive closures:

Basic Operations

We begin with a basic covering operation:	`cone()`
and then add its dual:	`support()`
followed by the transitive closures:	`closure(), star()`

Basic Operations

We begin with a basic covering operation:	`cone()`
and then add its dual:	`support()`
followed by the transitive closures:	`closure(), star()`
and finally lattice operations:	

Basic Operations

We begin with a basic covering operation:	`cone()`
and then add its dual:	`support()`
followed by the transitive closures:	`closure(), star()`
and finally lattice operations:	`meet(), join()`

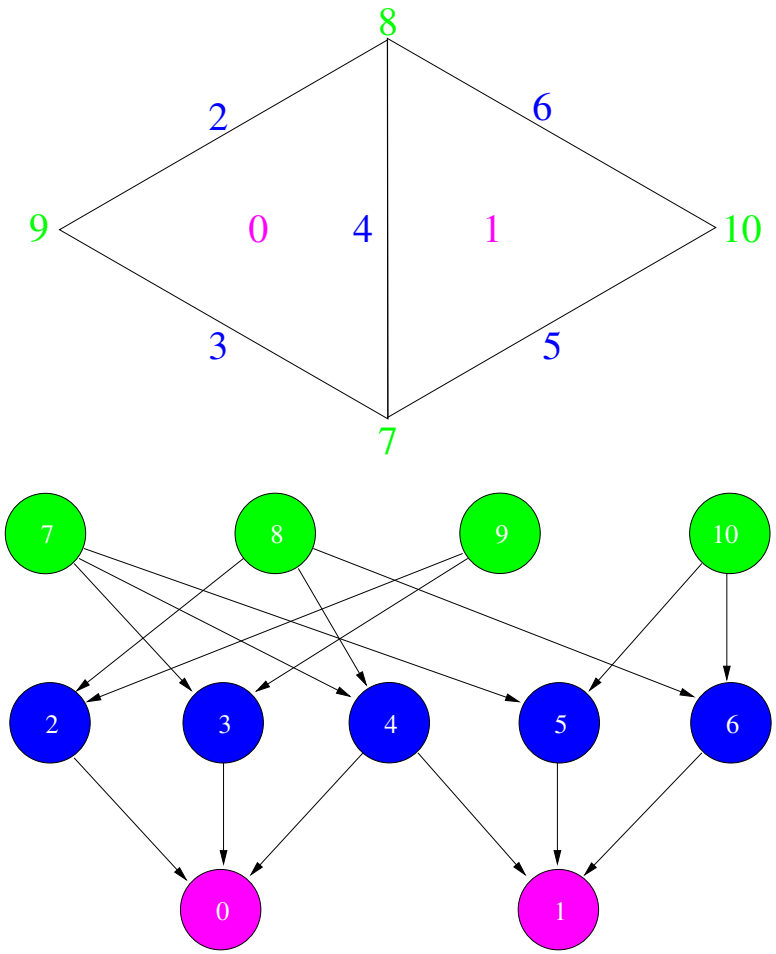
Sieve Definition

Definition

A Sieve consists of points, and arrows.
Each arrow connects a point to another which it covers.

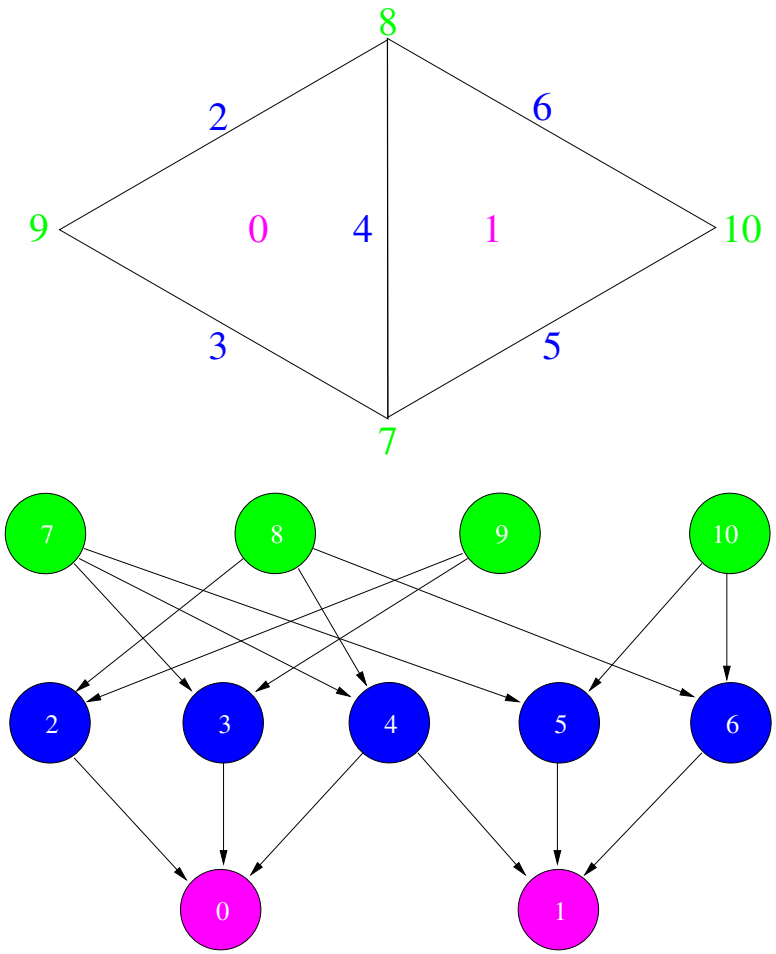
cone(p)	sequence of points which cover a given point p
closure(p)	transitive closure of cone
support(p)	sequence of points which are covered by a given point p
star(p)	transitive closure of support
meet(p,q)	minimal separator of closure(p) and closure(q)
join(p,q)	minimal separator of star(p) and star(q)

Doublet Mesh



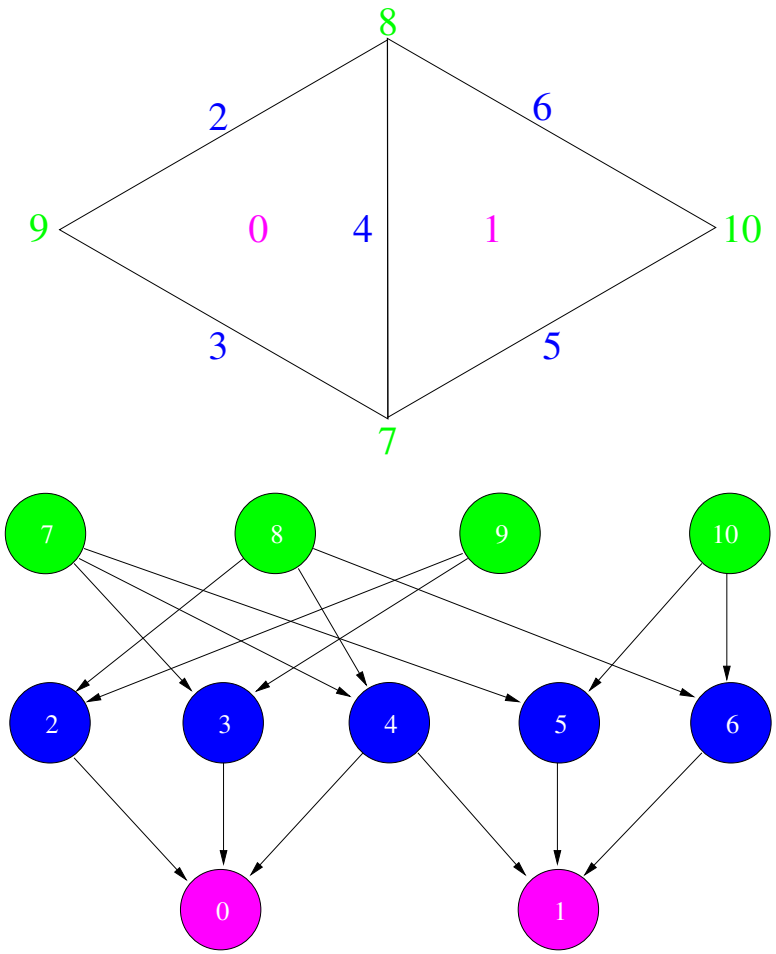
- Incidence/covering arrows
- $cone(0) = \{2, 3, 4\}$
- $support(7) = \{2, 3\}$

Doublet Mesh



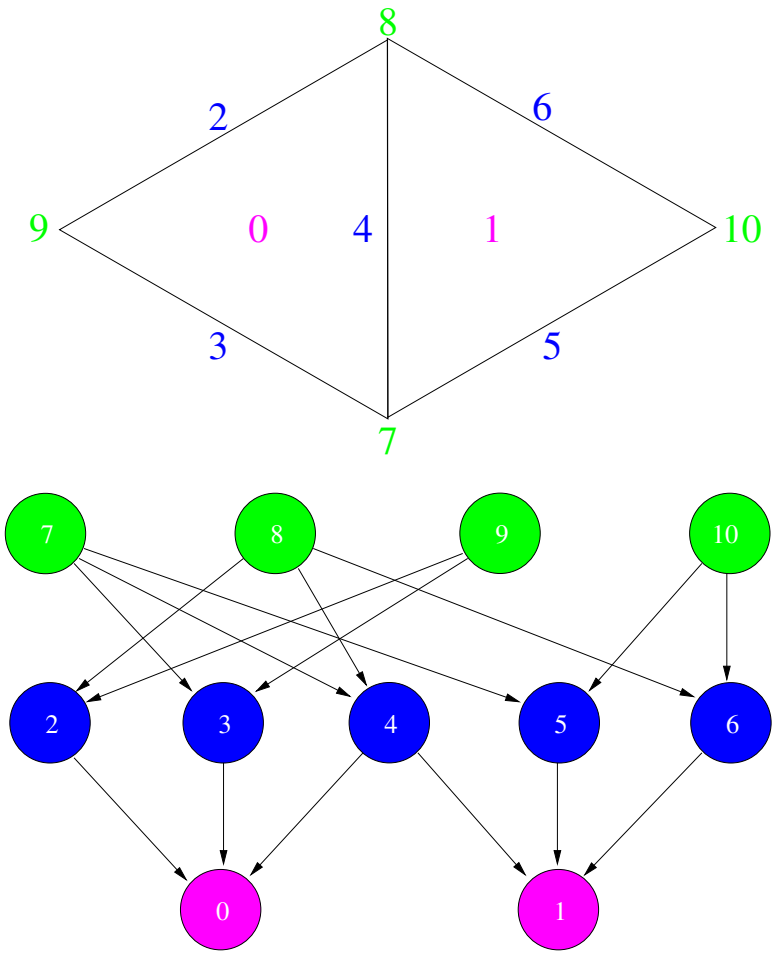
- Incidence/covering arrows
- $cone(0) = \{2, 3, 4\}$
- $support(7) = \{2, 3\}$

Doublet Mesh



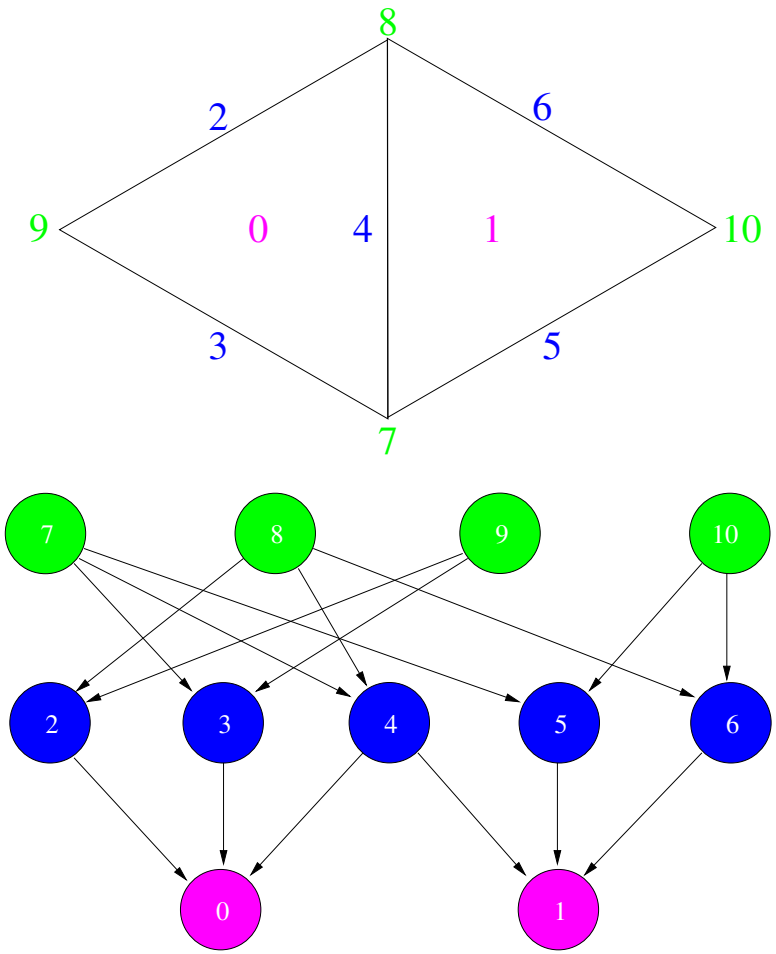
- Incidence/covering arrows
- $cone(0) = \{2, 3, 4\}$
- $support(7) = \{2, 3\}$

Doublet Mesh



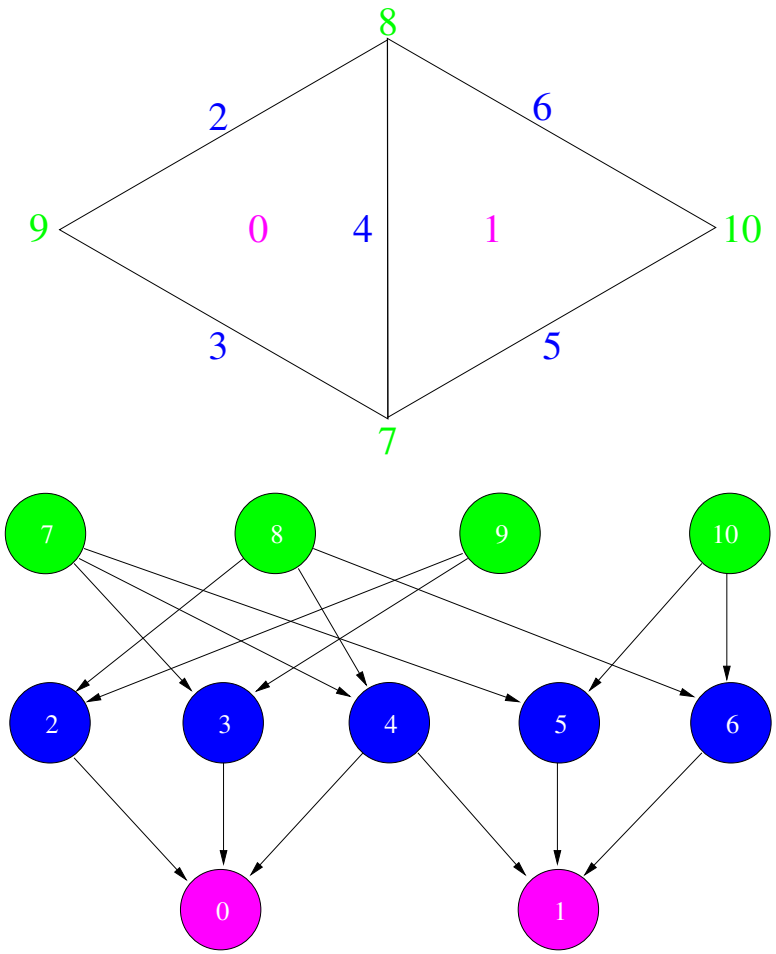
- Incidence/covering arrows
- $closure(0) = \{0, 2, 3, 4, 7, 8, 9\}$
- $star(7) = \{7, 2, 3, 0\}$

Doublet Mesh



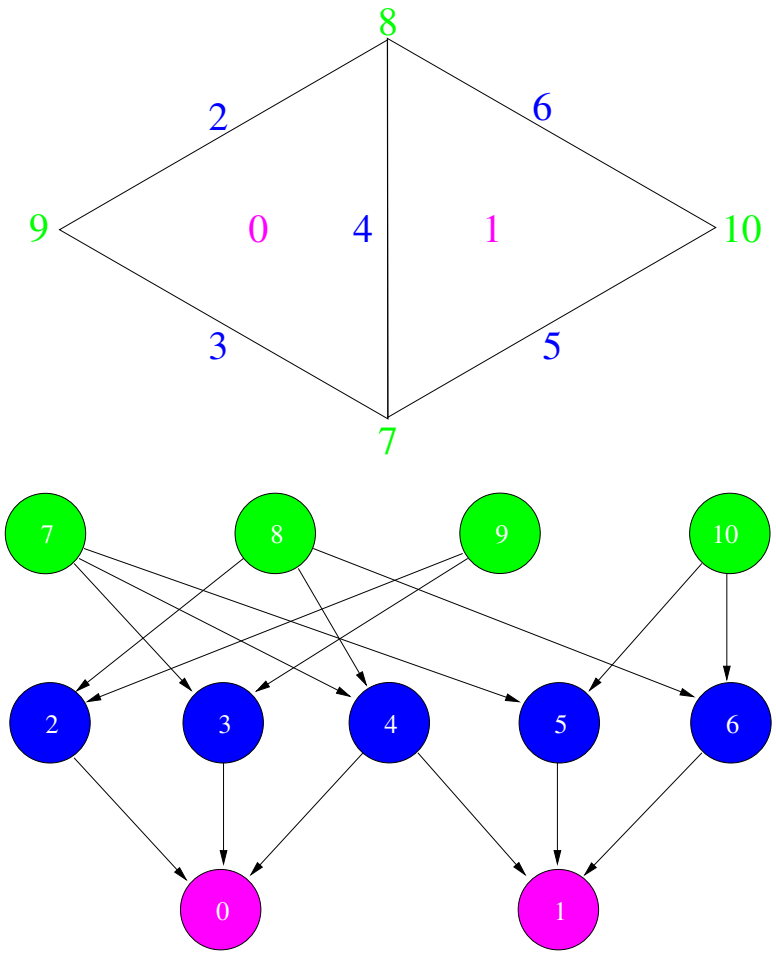
- Incidence/covering arrows
- $closure(0) = \{0, 2, 3, 4, 7, 8, 9\}$
- $star(7) = \{7, 2, 3, 0\}$

Doublet Mesh



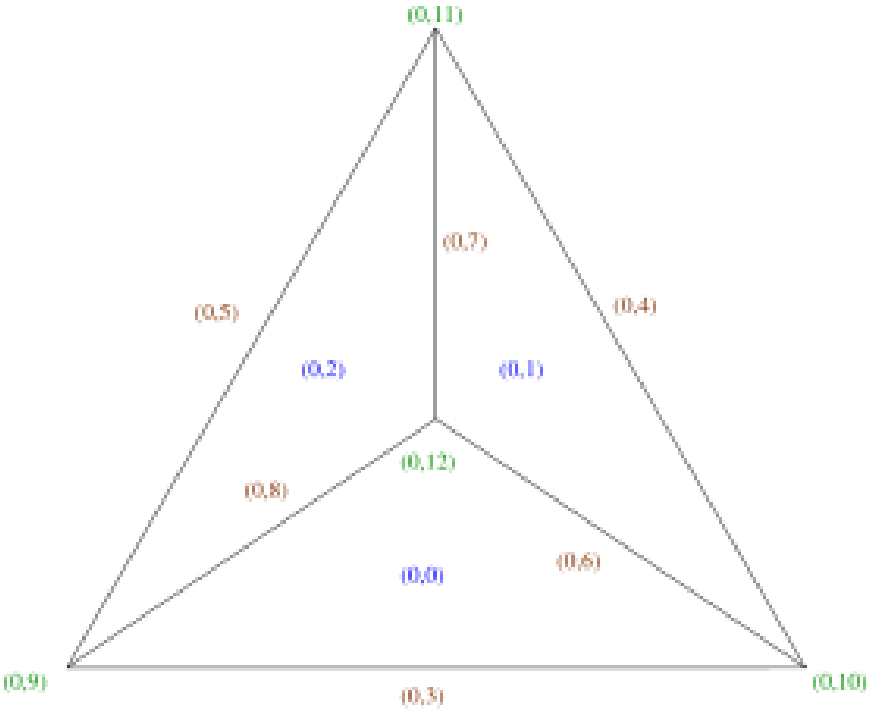
- Incidence/covering arrows
- $meet(0, 1) = \{4\}$
- $join(8, 9) = \{4\}$

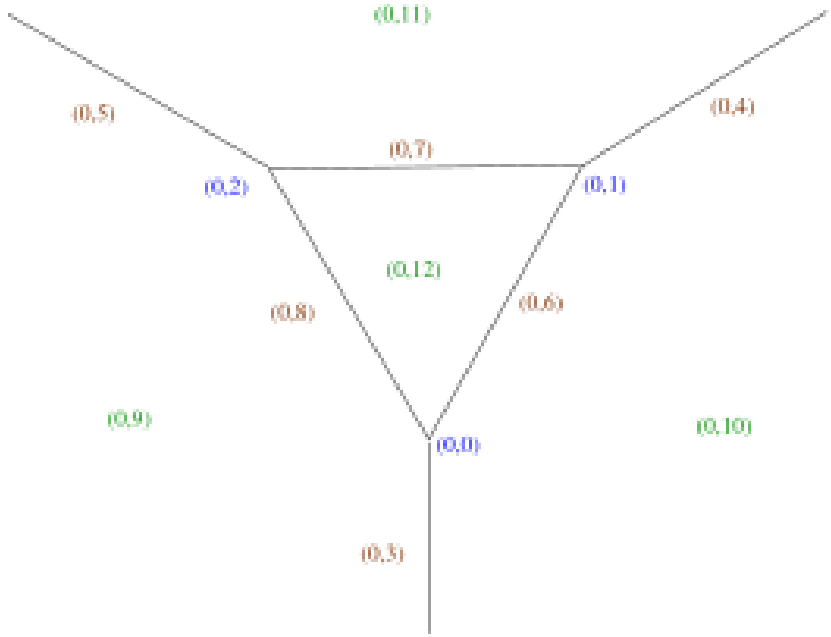
Doublet Mesh

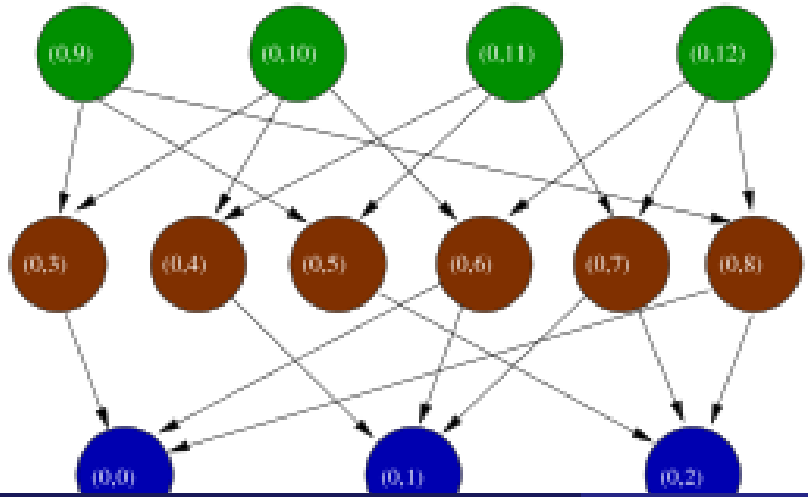


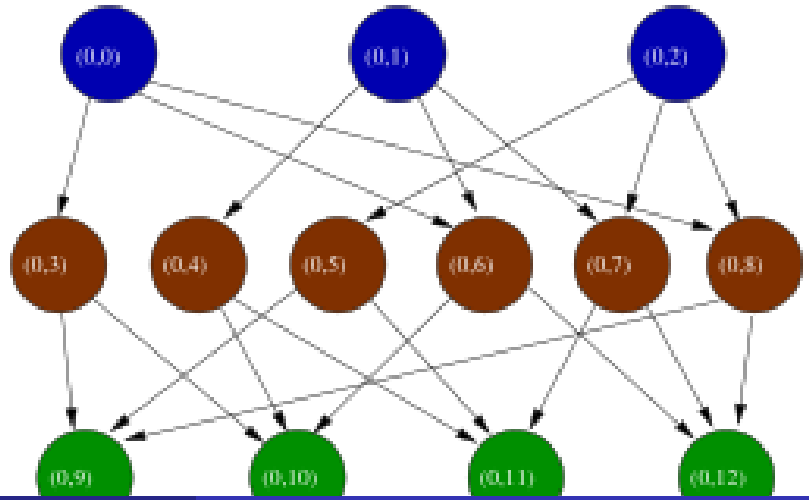
- Incidence/covering arrows
- $meet(0, 1) = \{4\}$
- $join(8, 9) = \{4\}$

The Mesh Dual









Outline

Mesh Distribution

Distributing a mesh means

- distributing the topology (Sieve)
- distributing data (Section)

However, a Sieve can be interpreted as a Section of `cone()`s!

Mesh Distribution

Distributing a mesh means

- distributing the topology (Sieve)
- distributing data (Section)

However, a Sieve can be interpreted as a Section of `cone()`s!

Mesh Distribution

Distributing a mesh means

- distributing the topology (Sieve)
- distributing data (Section)

However, a Sieve can be interpreted as a Section of `cone()`s!

Mesh Distribution

Distributing a mesh means

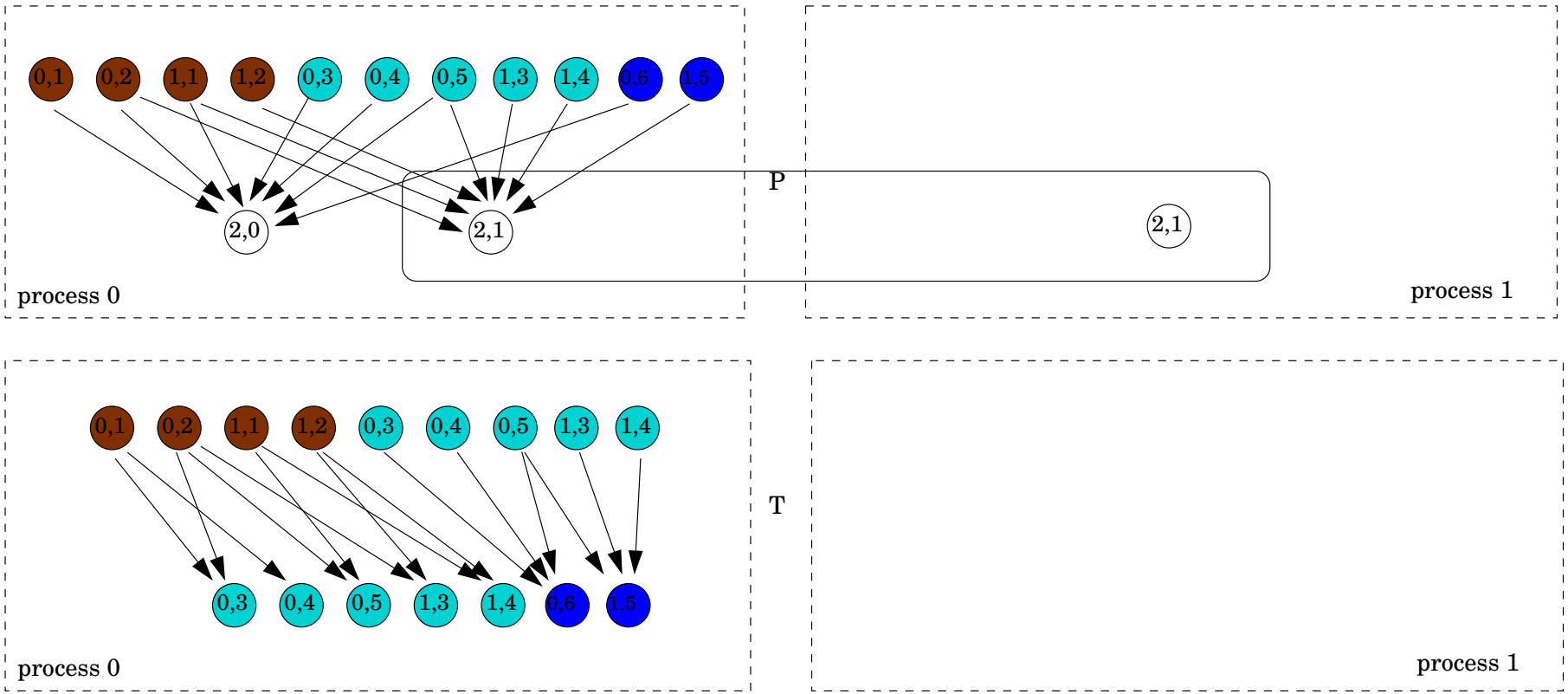
- distributing the topology (Sieve)
- distributing data (Section)

However, a Sieve can be interpreted as a Section of `cone()`s!

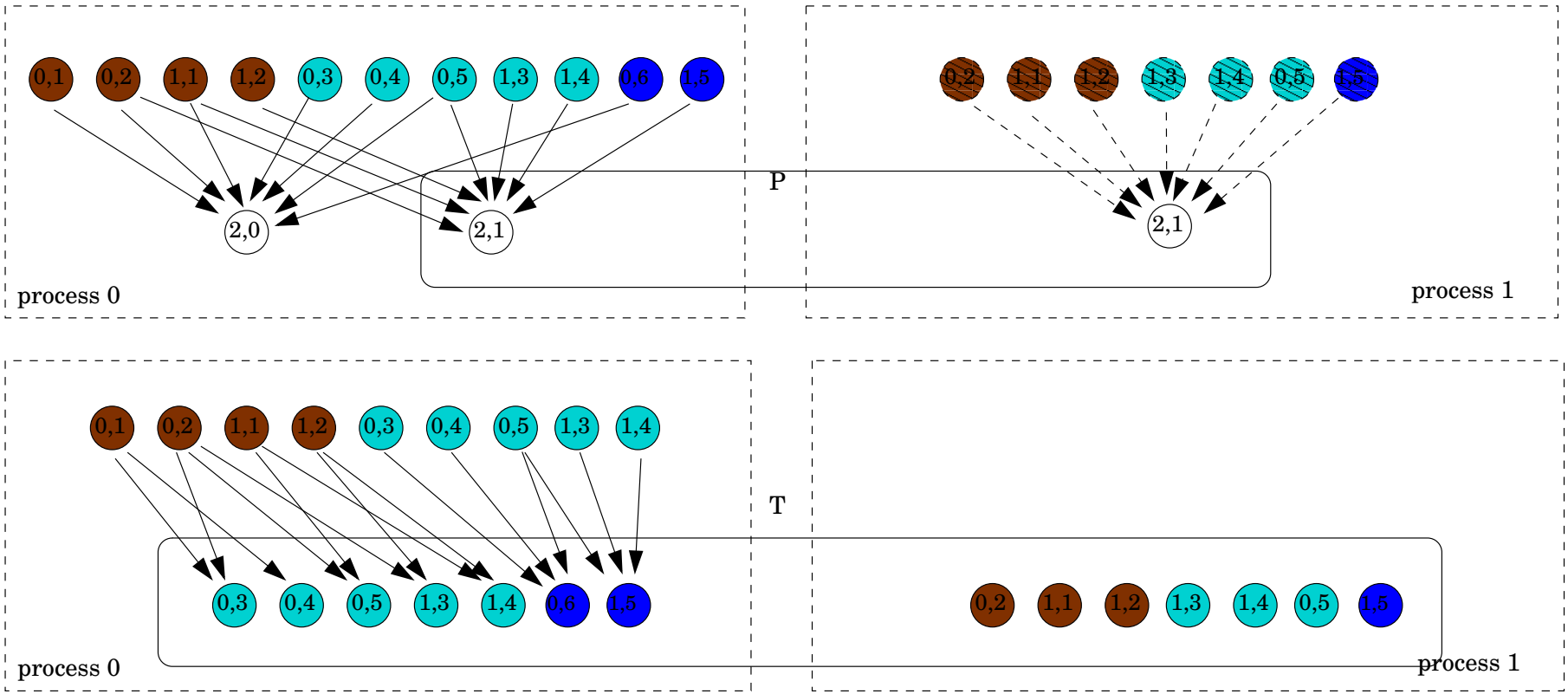
Mesh Partition

- 3rd party packages construct a vertex partition
- For FEM, partition dual graph vertices
- For FVM, construct hyperpgraph dual with faces as vertices
- Assign `closure(v)` and `star(v)` to same partition

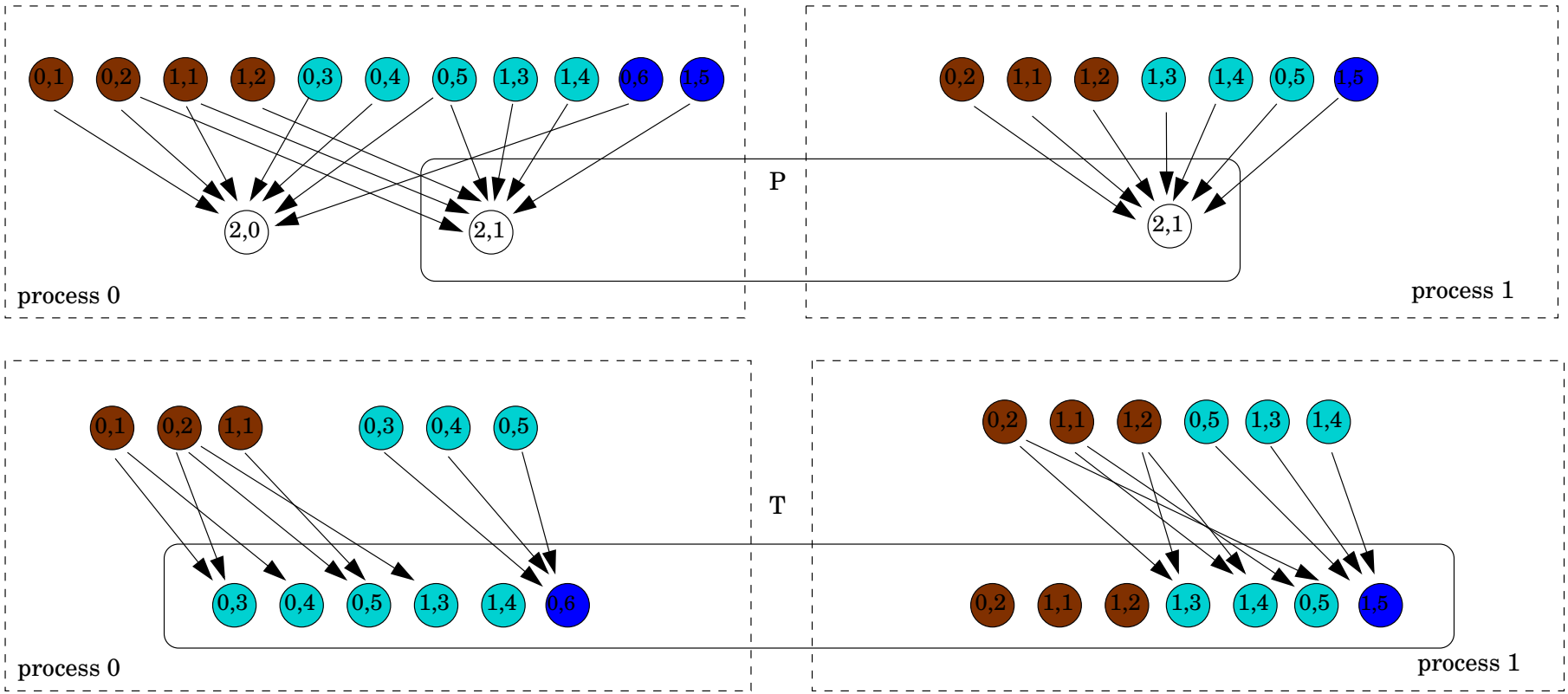
Doublet Mesh Distribution



Doublet Mesh Distribution



Doublet Mesh Distribution



Section Distribution

Section distribution consists of

- Creation of the local Section
- Distribution of the Atlas (layout Section)
- Completion of the Section

Sieve Distribution

1. Construct local mesh from partition
2. Construct partition overlap
3. `Complete()` the partition section
   - This distributes the cells
4. Update Overlap with new points
5. `Complete()` the cone section
   - This distributes the remaining sieve points
6. Update local Sieves

Sieve Distribution

1. Construct local mesh from partition
2. Construct partition overlap
3. `Complete()` the partition section
   - This distributes the cells
4. Update Overlap with new points
5. `Complete()` the cone section
   - This distributes the remaining sieve points
6. Update local Sieves

Sieve Distribution

1. Construct local mesh from partition
2. Construct partition overlap
3. `Complete()` the partition section
   - This distributes the cells
4. Update Overlap with new points
5. `Complete()` the cone section
   - This distributes the remaining sieve points
6. Update local Sieves

Sieve Distribution

1. Construct local mesh from partition
2. Construct partition overlap
3. `Complete()` the partition section
   - This distributes the cells
4. Update Overlap with new points
5. `Complete()` the cone section
   - This distributes the remaining sieve points
6. Update local Sieves

Sieve Distribution

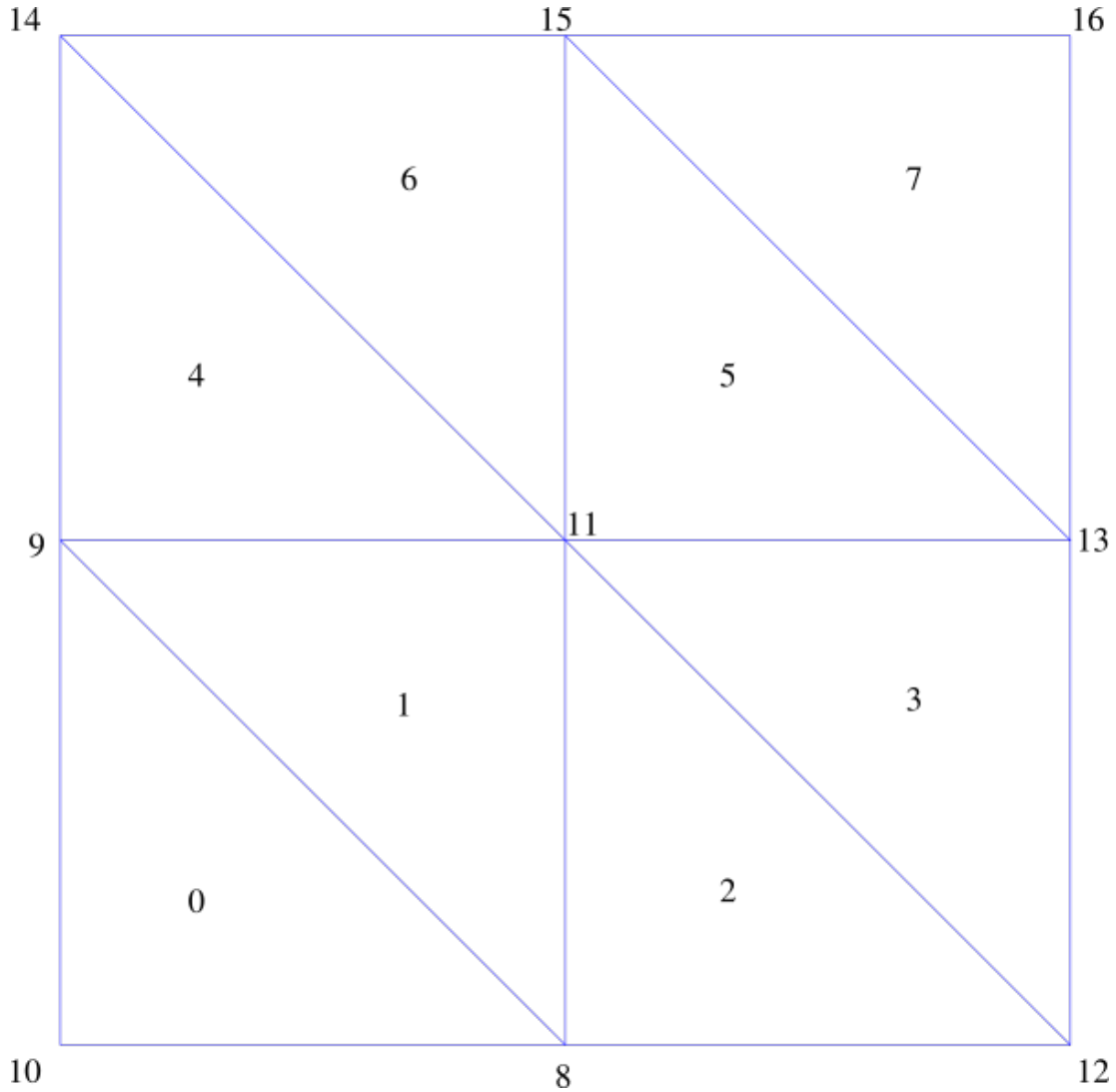
1. Construct local mesh from partition
2. Construct partition overlap
3. `Complete()` the partition section
   - This distributes the cells
4. Update Overlap with new points
5. `Complete()` the cone section
   - This distributes the remaining sieve points
6. Update local Sieves

Sieve Distribution

1. Construct local mesh from partition
2. Construct partition overlap
3. `Complete()` the partition section
   - This distributes the cells
4. Update Overlap with new points
5. `Complete()` the cone section
   - This distributes the remaining sieve points
6. Update local Sieves

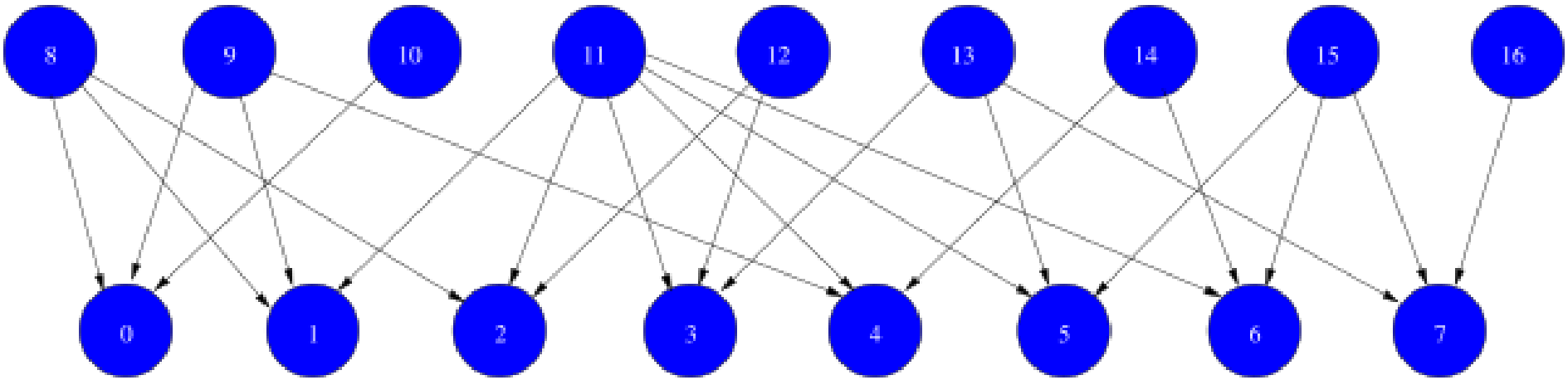
2D Example

A simple triangular mesh



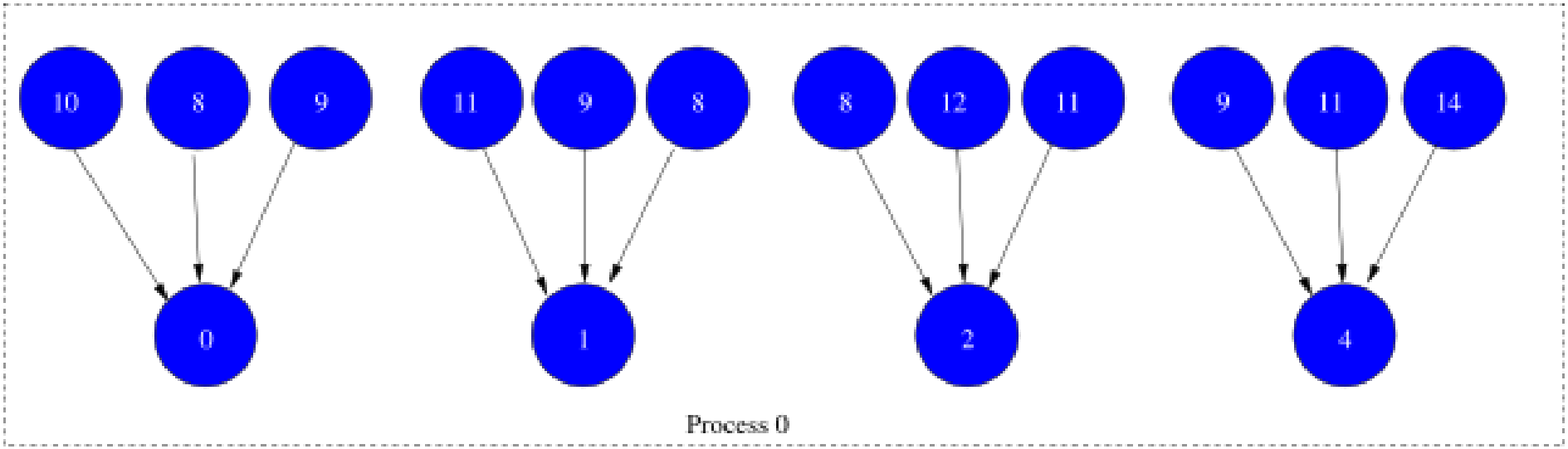
2D Example

Sieve for the mesh



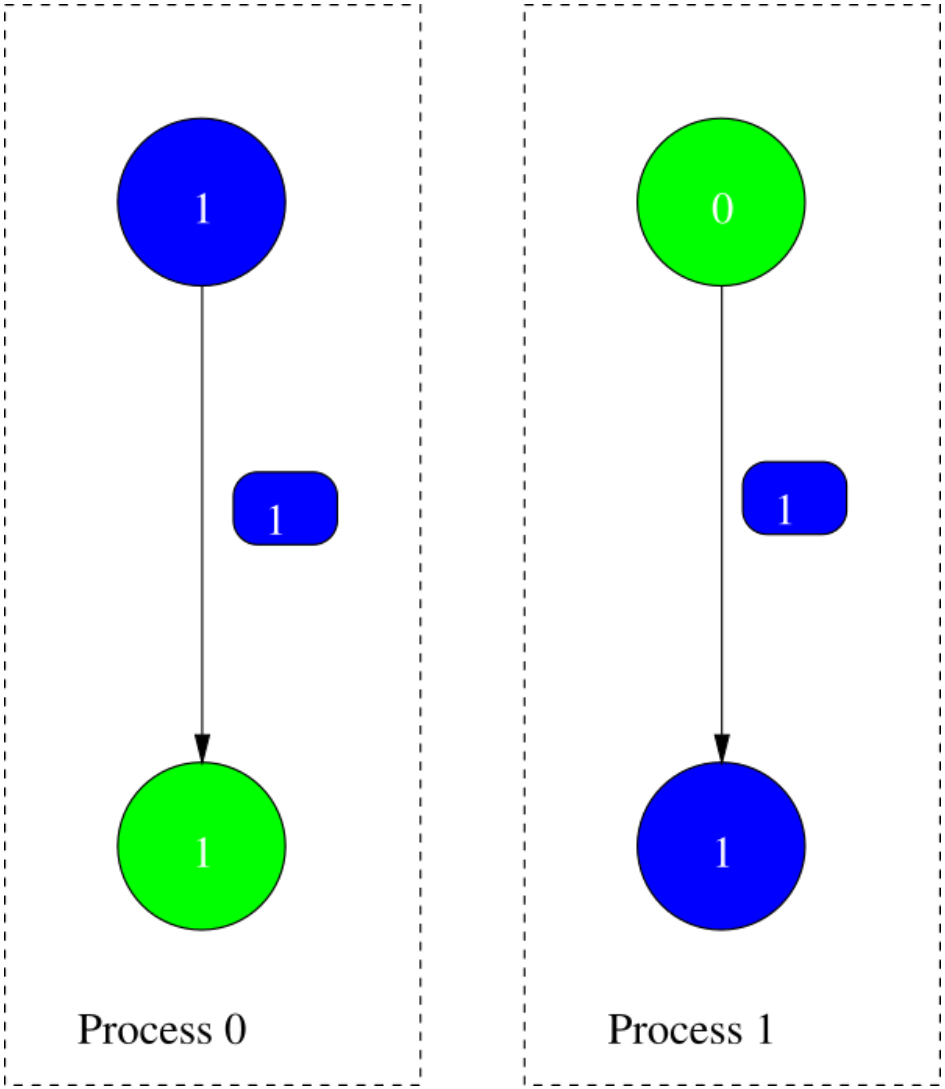
2D Example

Local sieve on process 0



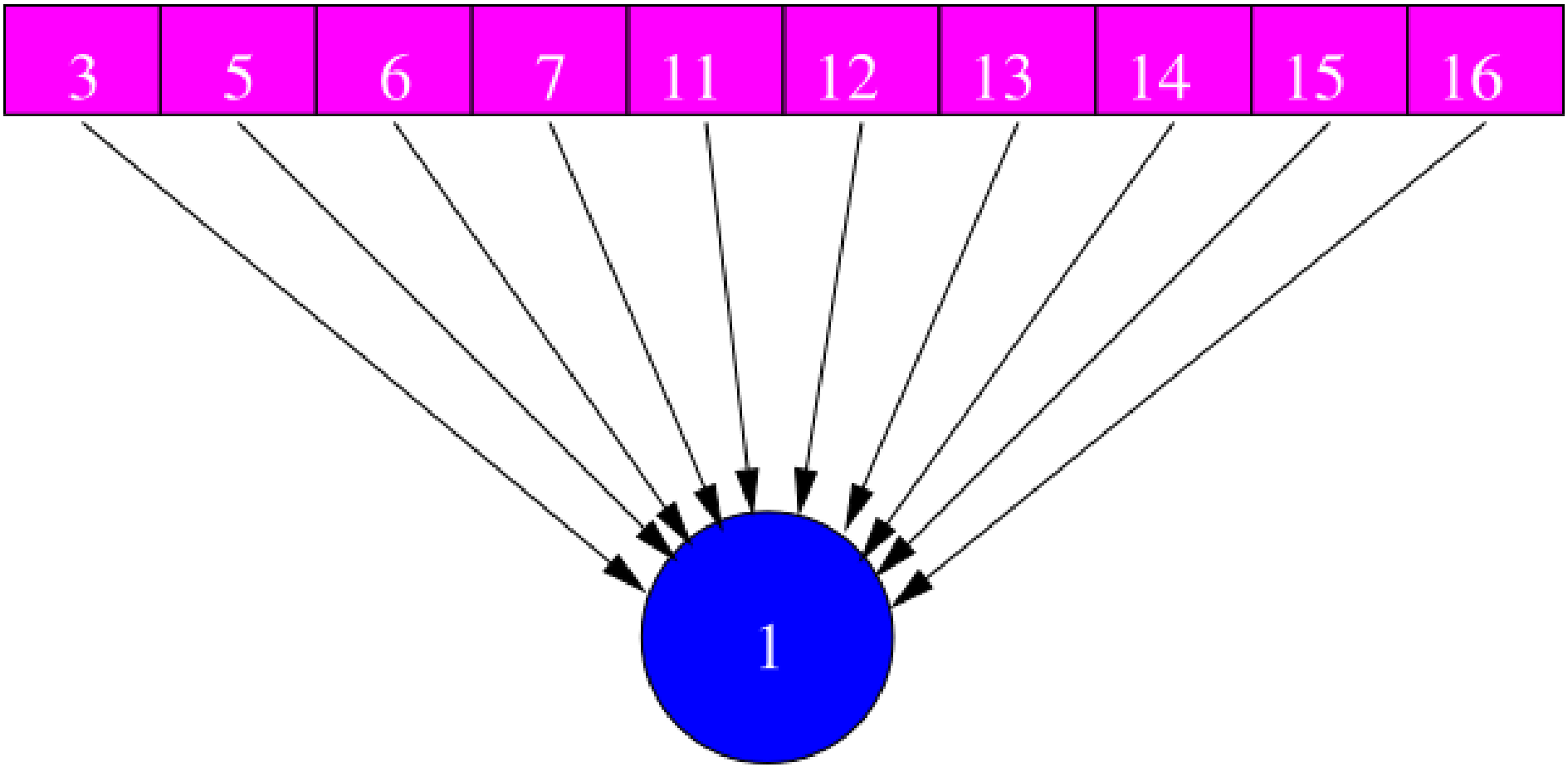
2D Example

Partition Overlap



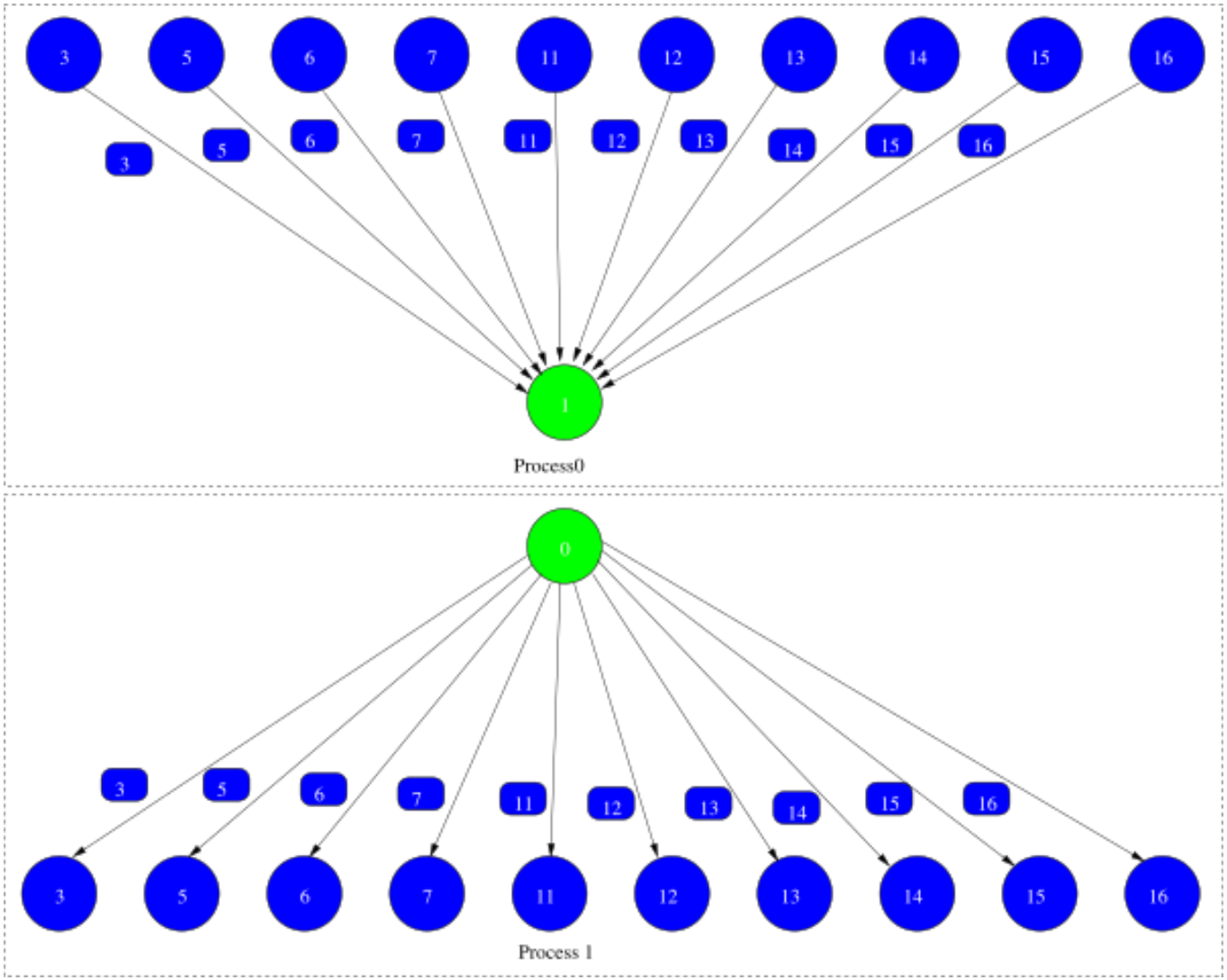
2D Example

Partition Section



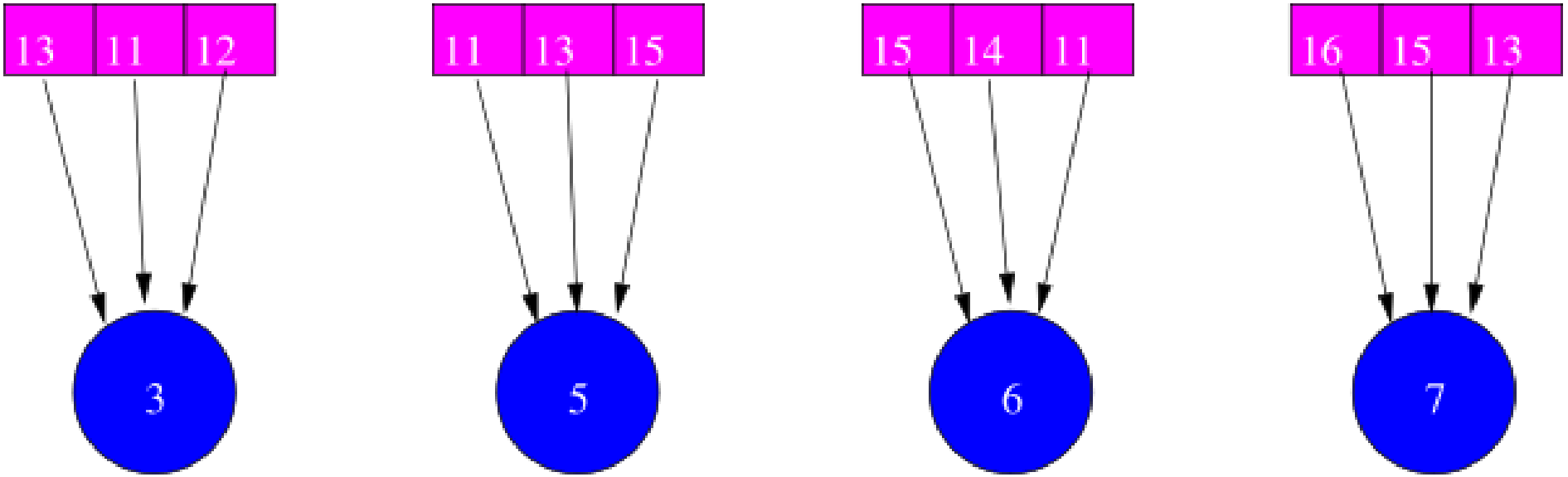
2D Example

Updated Sieve Overlap



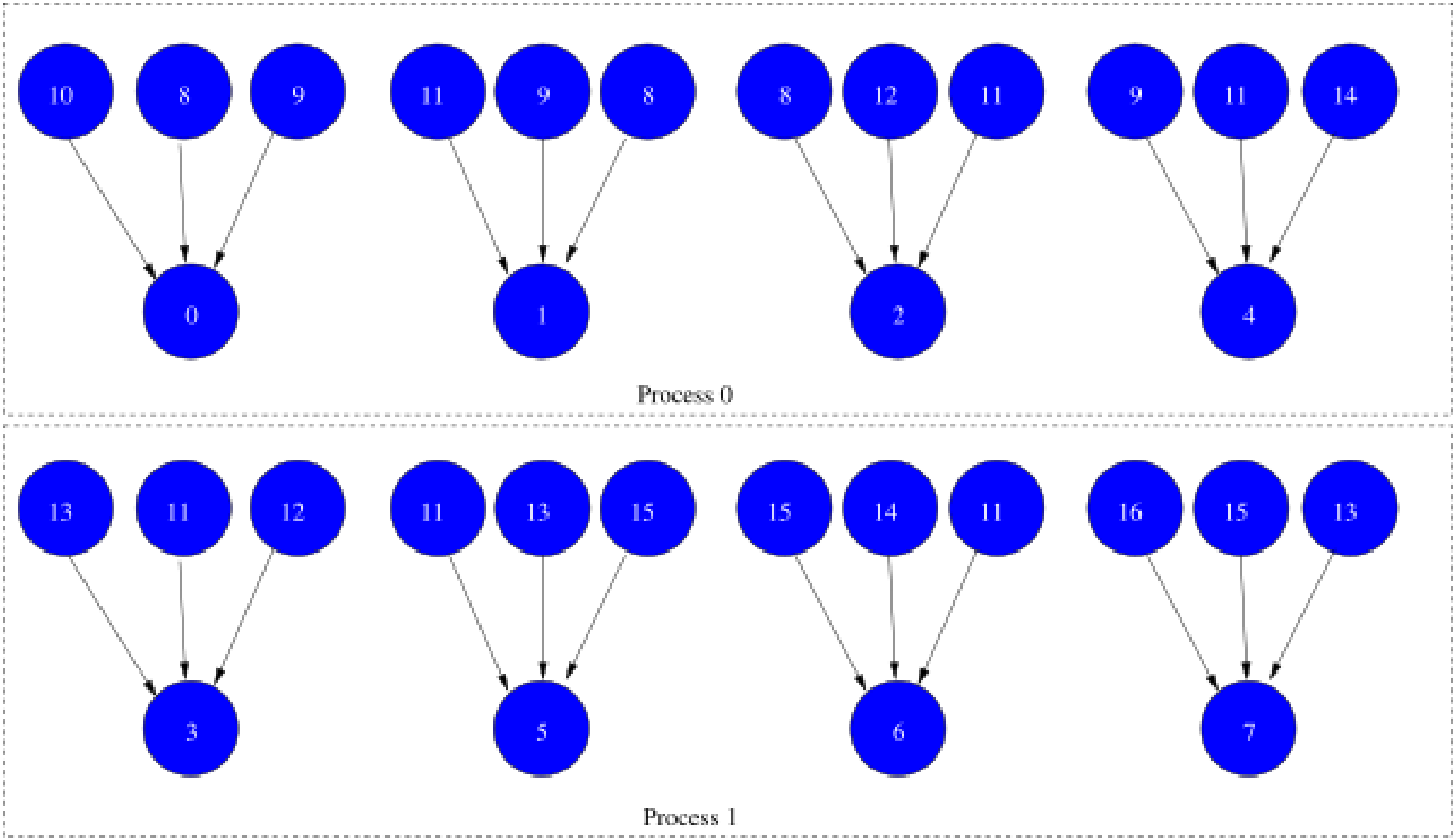
2D Example

Cone Section



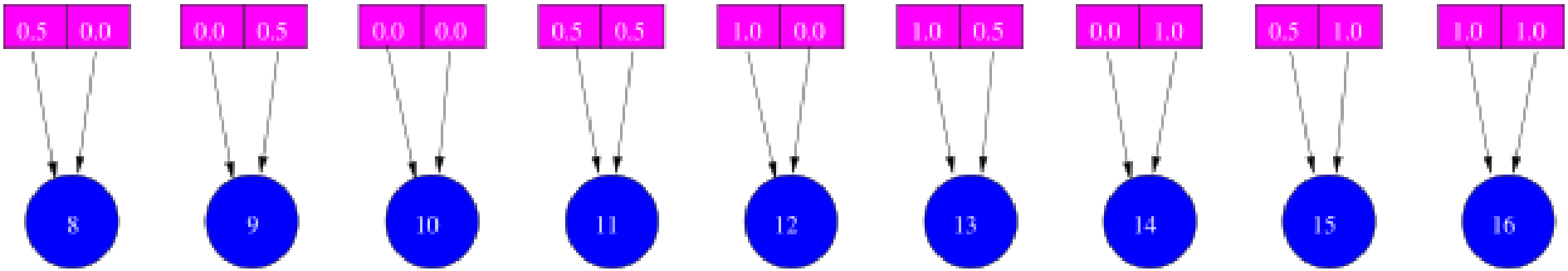
2D Example

Distributed Sieve



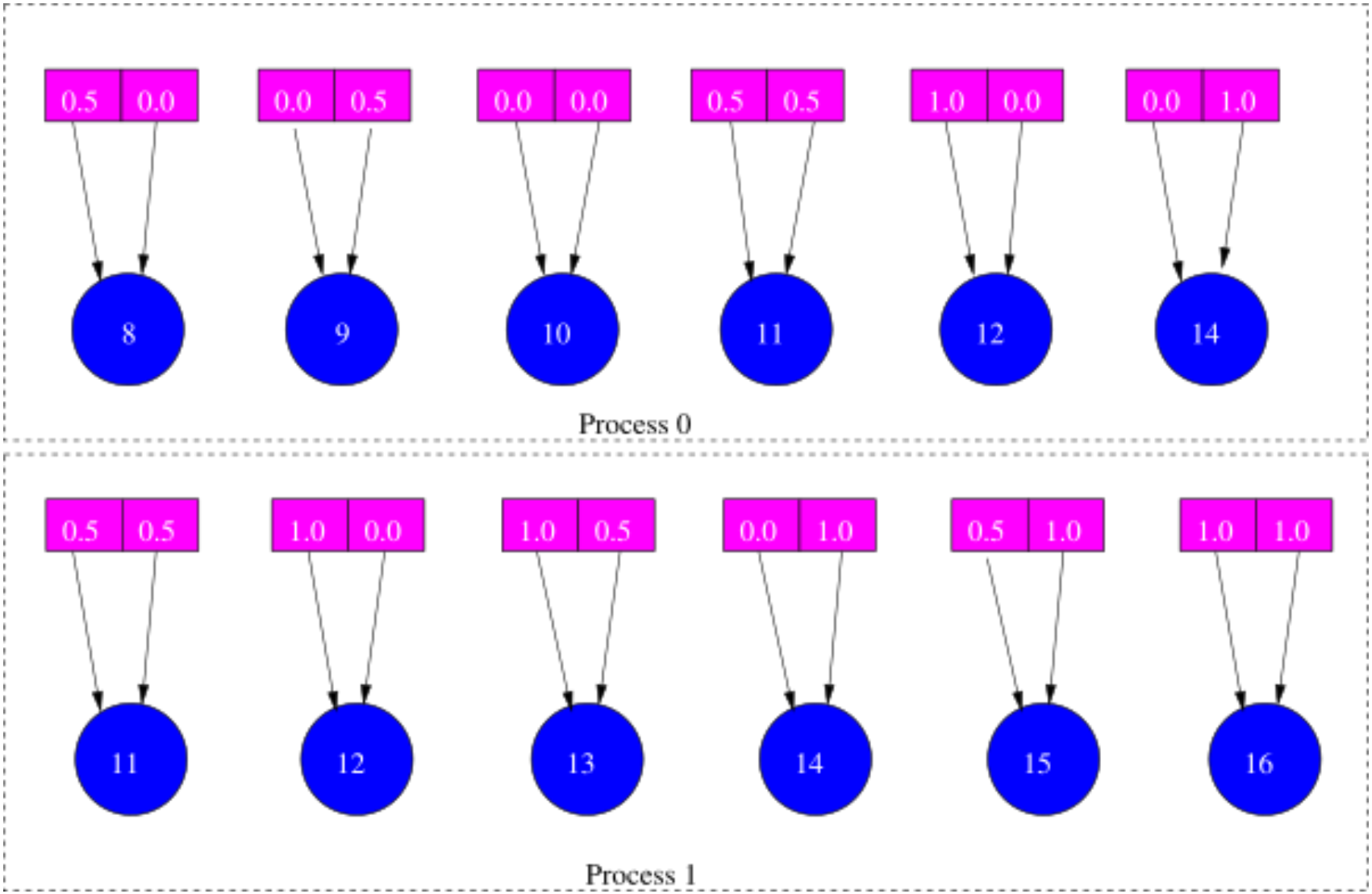
2D Example

Coordinate Section



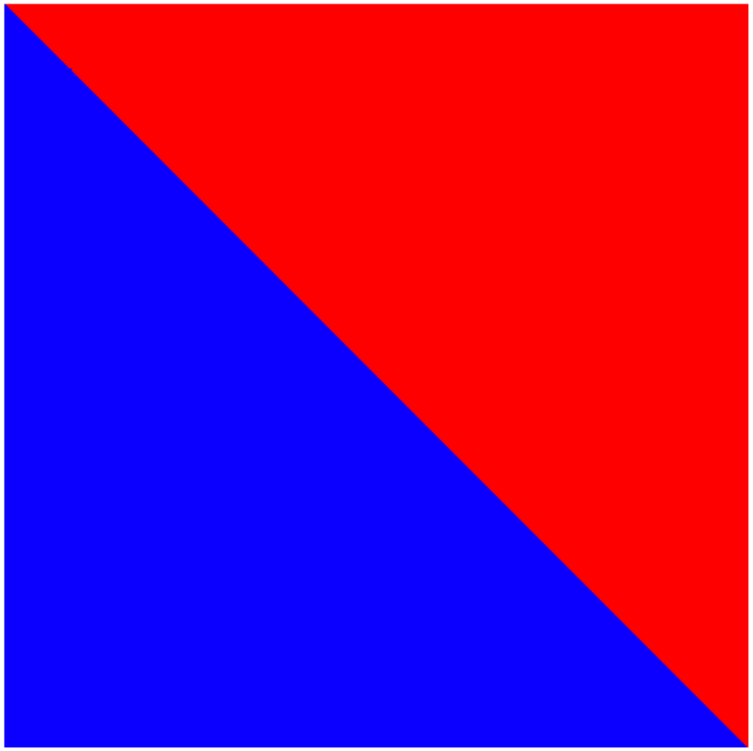
2D Example

Distributed Coordinate Section



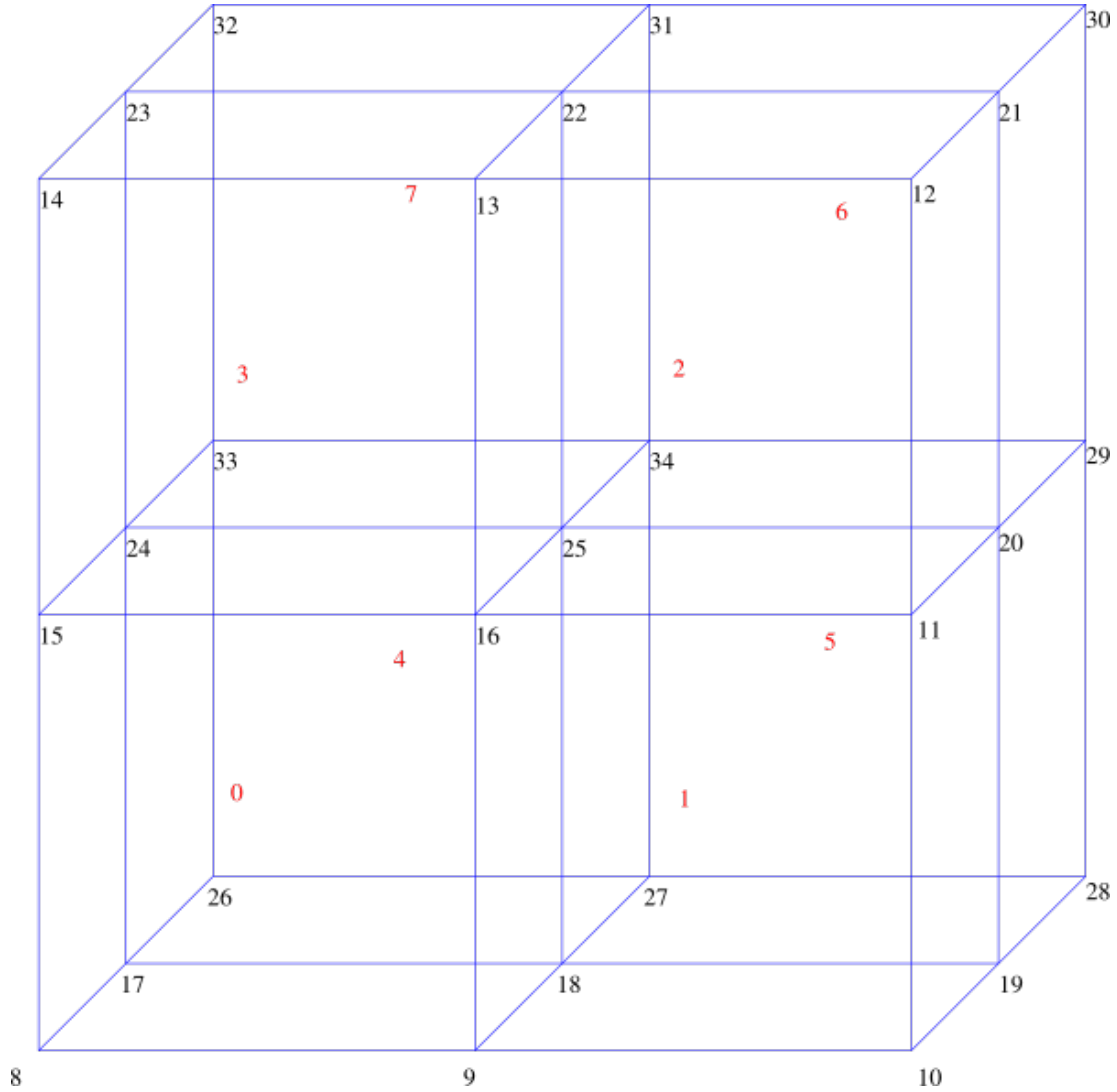
2D Example

Distributed Mesh



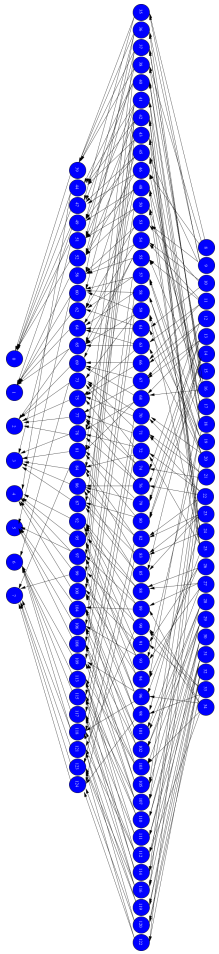
3D Example

A simple hexahedral mesh



3D Example

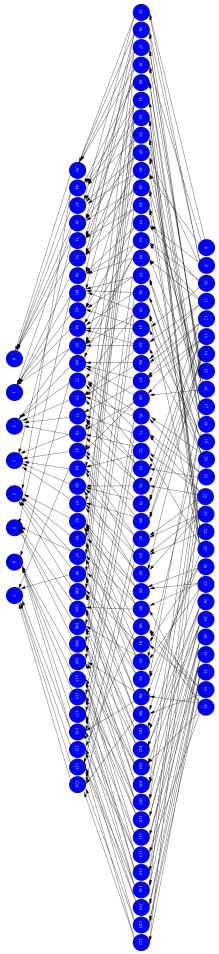
Sieve for the mesh



Its complicated!

3D Example

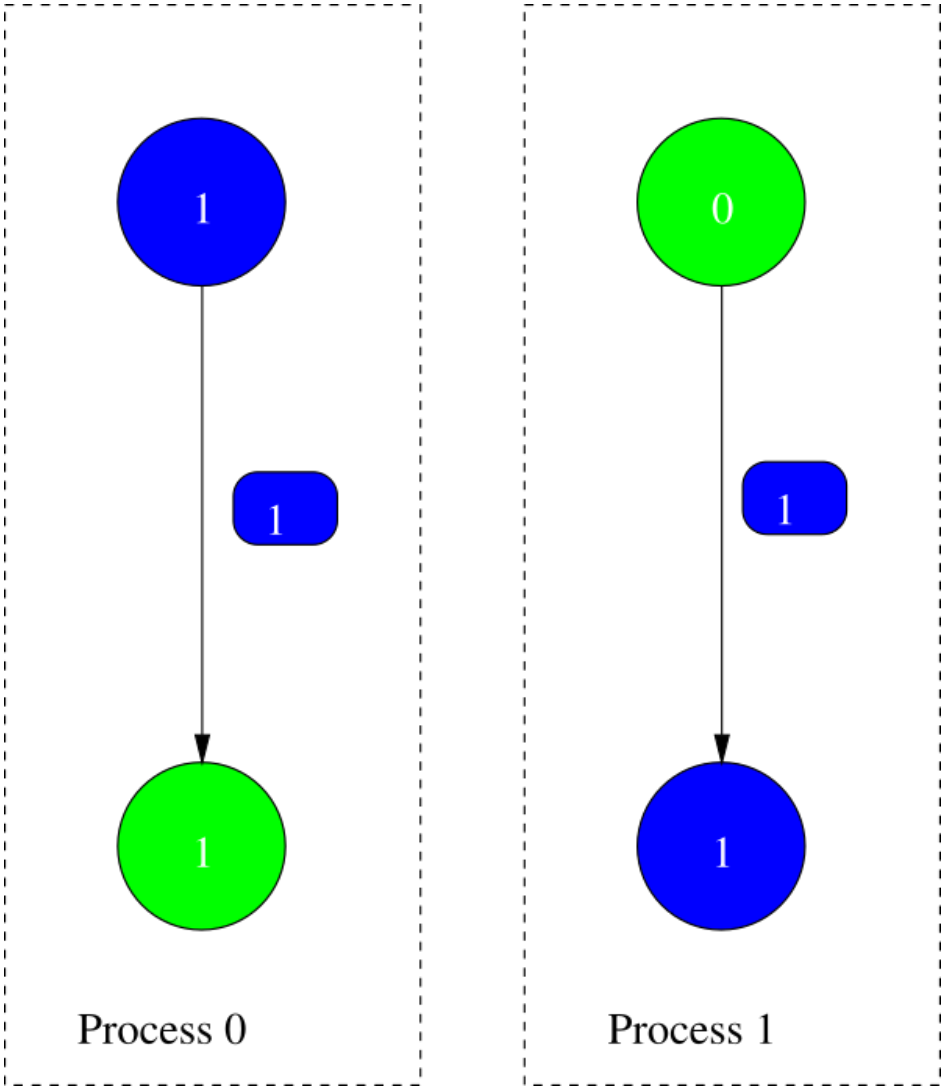
Sieve for the mesh



Its complicated!

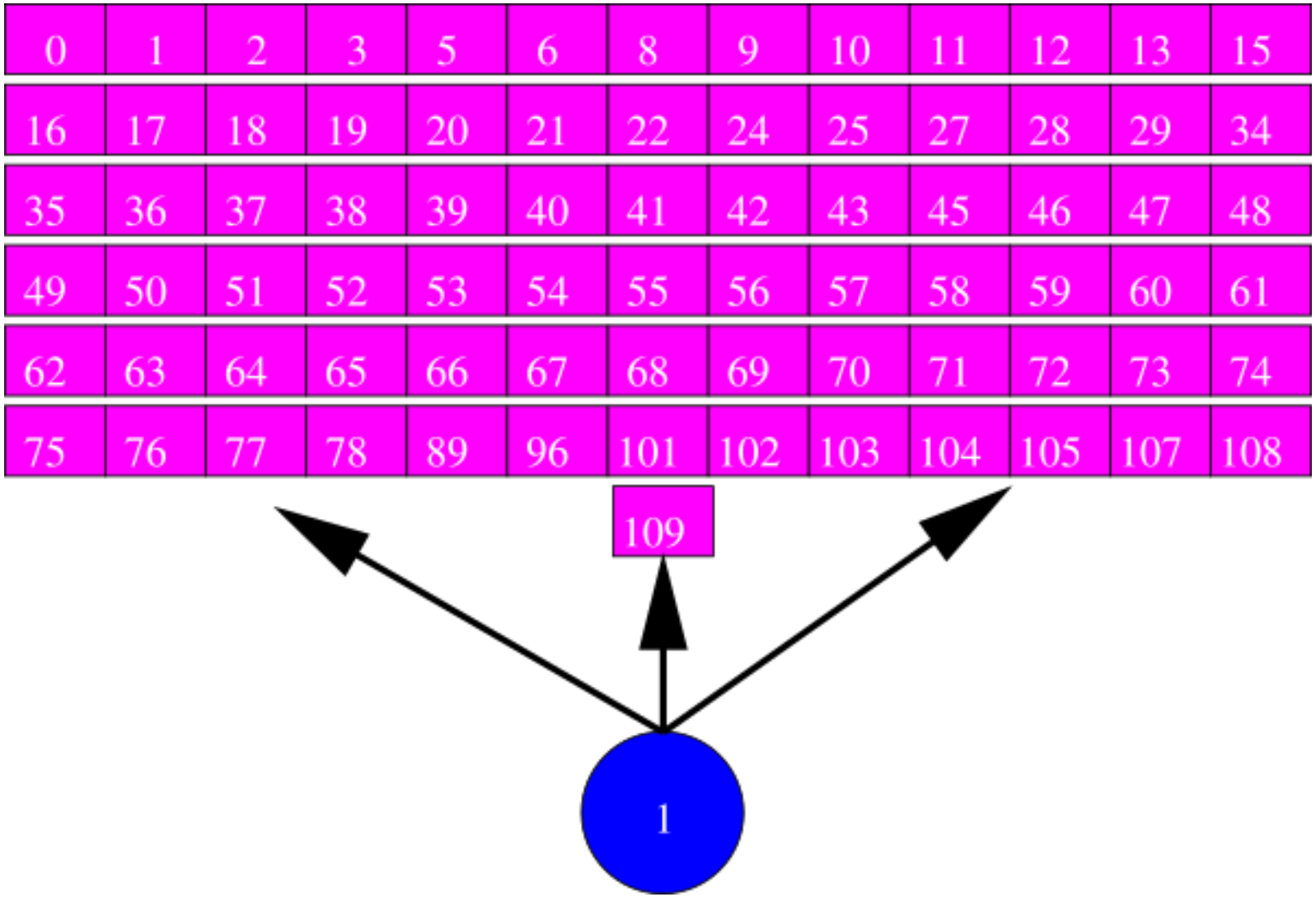
3D Example

Partition Overlap



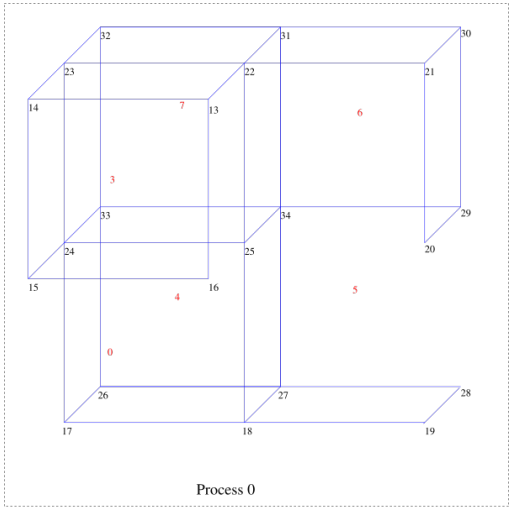
3D Example

Partition Section



3D Example

Distributed Mesh



Notice cells are ghosted

Outline

Sections

Sections associate data to submeshes

- Name comes from section of a fiber bundle
  - Generalizes linear algebra paradigm
- Define `restrict()`,`update()`
- Define `complete()`
- Assembly routines take a `Sieve` and several `Section`s
  - This is called a `Bundle`

Basic Operations

We begin with a simple mapping operation:

Basic Operations

We begin with a simple mapping operation: `restrictPoint()`

Basic Operations

We begin with a simple mapping operation: `restrictPoint()`
and then add its converse:

Basic Operations

We begin with a simple mapping operation: `restrictPoint()`
and then add its converse: `updatePoint()`

Basic Operations

We begin with a simple mapping operation: `restrictPoint()`
and then add its converse: `updatePoint()`
followed by topological versions:
which appear as dual to covering,

Basic Operations

We begin with a simple mapping operation:	`restrictPoint()`
and then add its converse:	`updatePoint()`
followed by topological versions:	`restrictClosure()`
which appear as dual to covering,	`updateClosure()`

Basic Operations

We begin with a simple mapping operation:	`restrictPoint()`
and then add its converse:	`updatePoint()`
followed by topological versions:	`restrictClosure()`
which appear as dual to covering,	`updateClosure()`
and finally a consistency operation:	

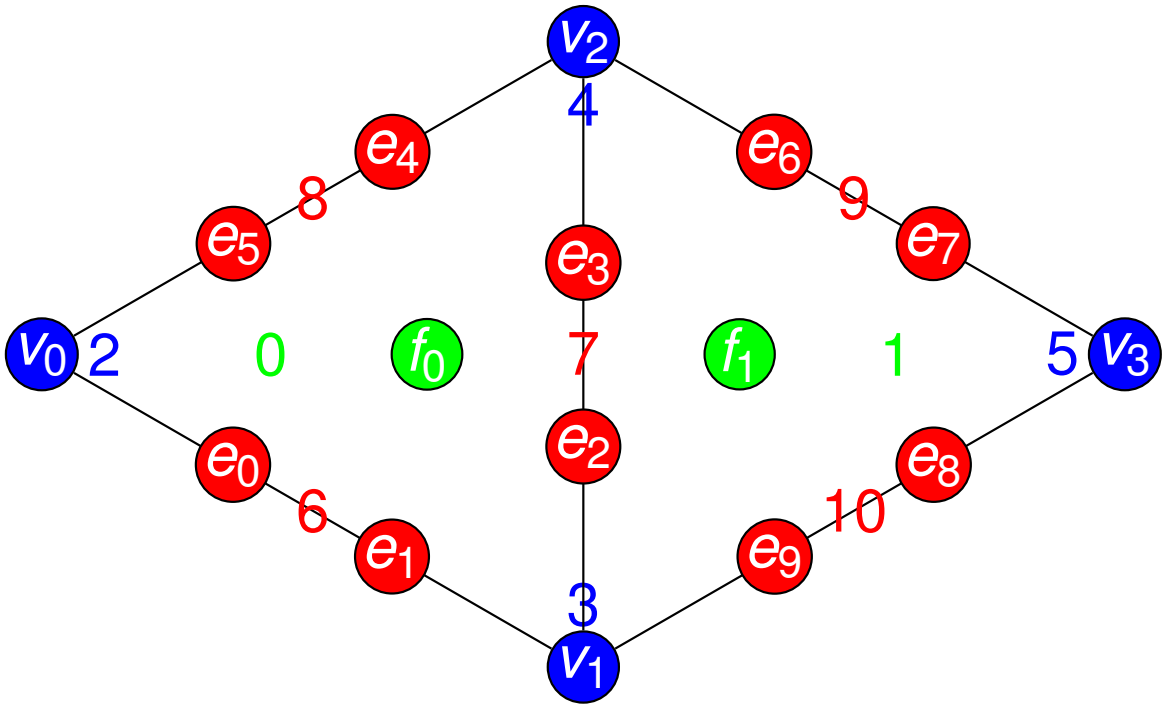
Basic Operations

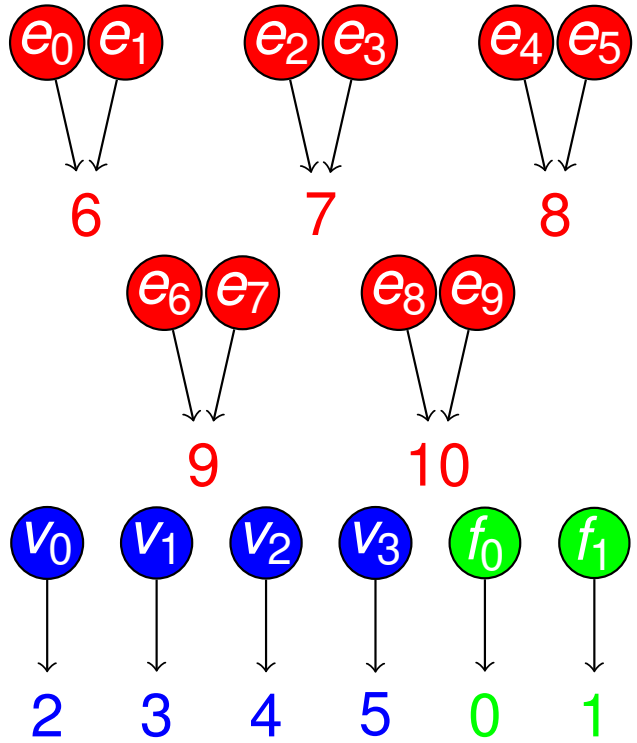
We begin with a simple mapping operation:	`restrictPoint()`
and then add its converse:	`updatePoint()`
followed by topological versions:	`restrictClosure()`
which appear as dual to covering,	`updateClosure()`
and finally a consistency operation:	`complete()`

Duality

- Need picture of sieve (graph) <–> mesh (picture) maybe doublet
- Show both traversals (closure and restriction), perhaps an animated FEM integral

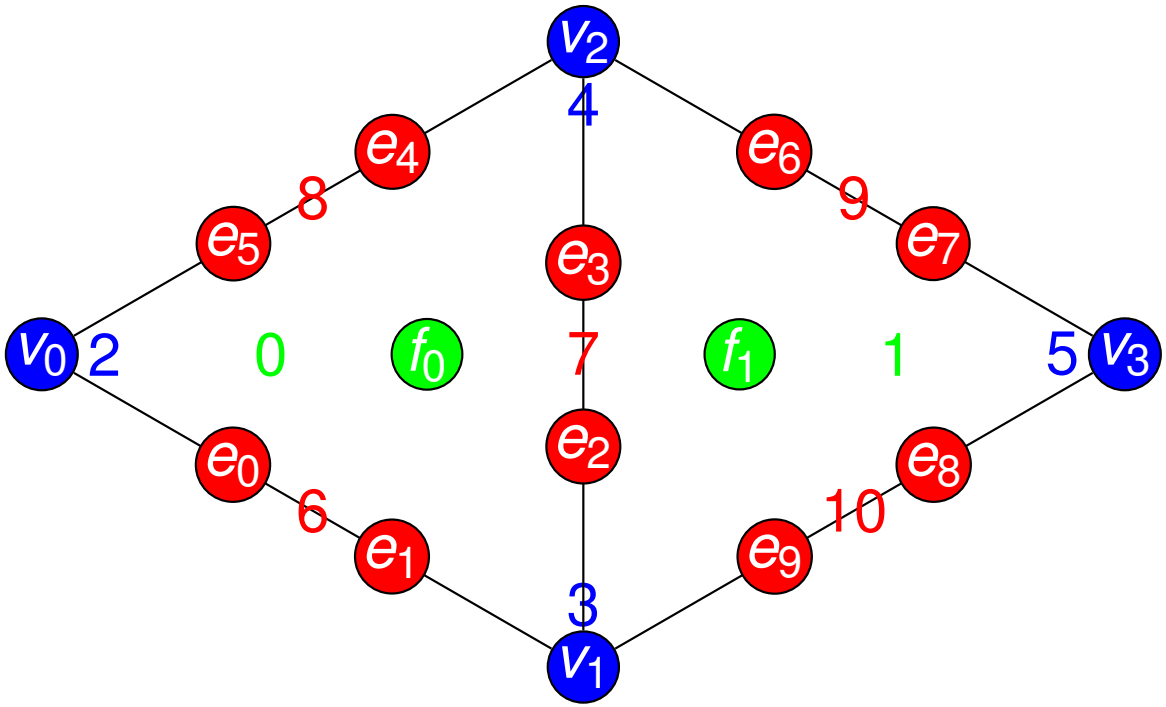
Doublet Section

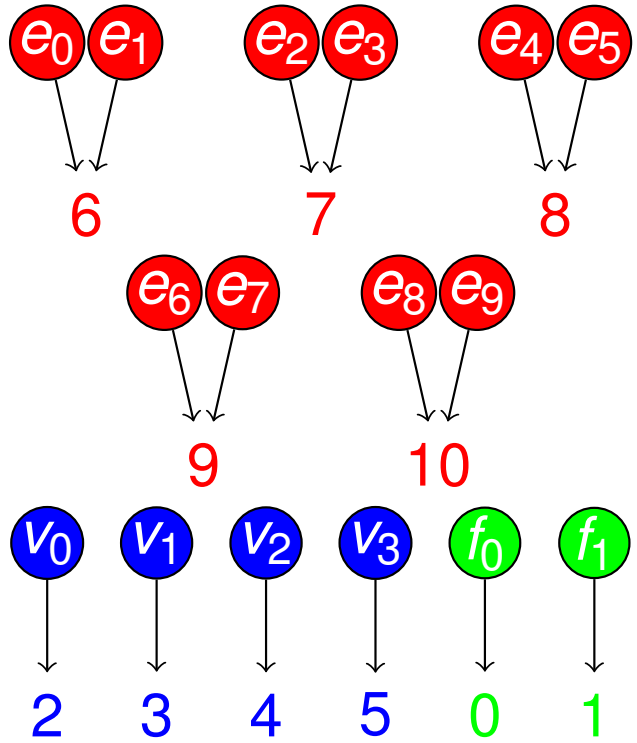




- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

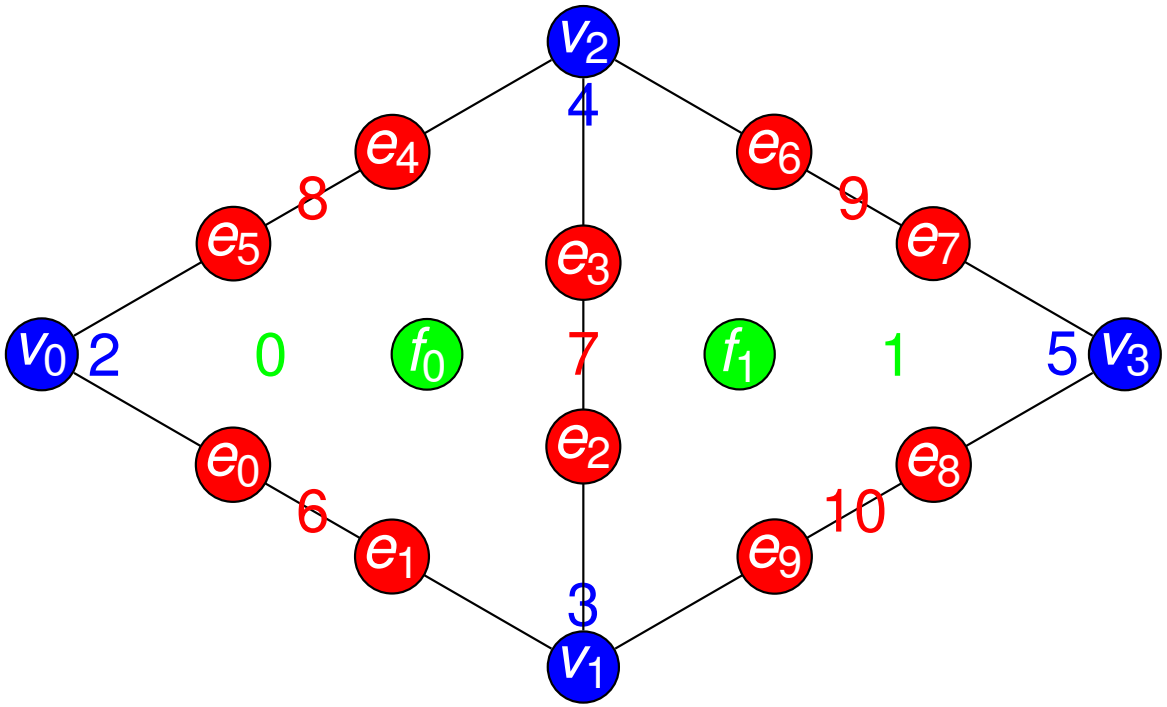
Doublet Section

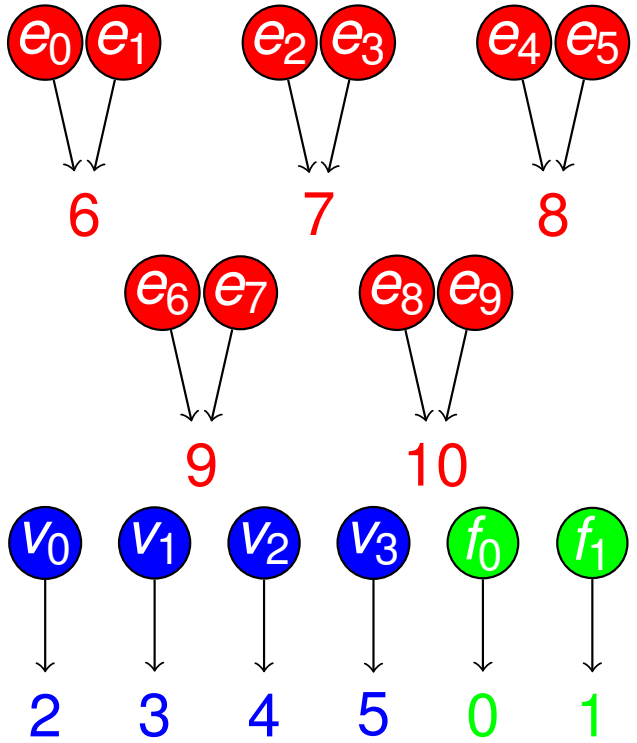




- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

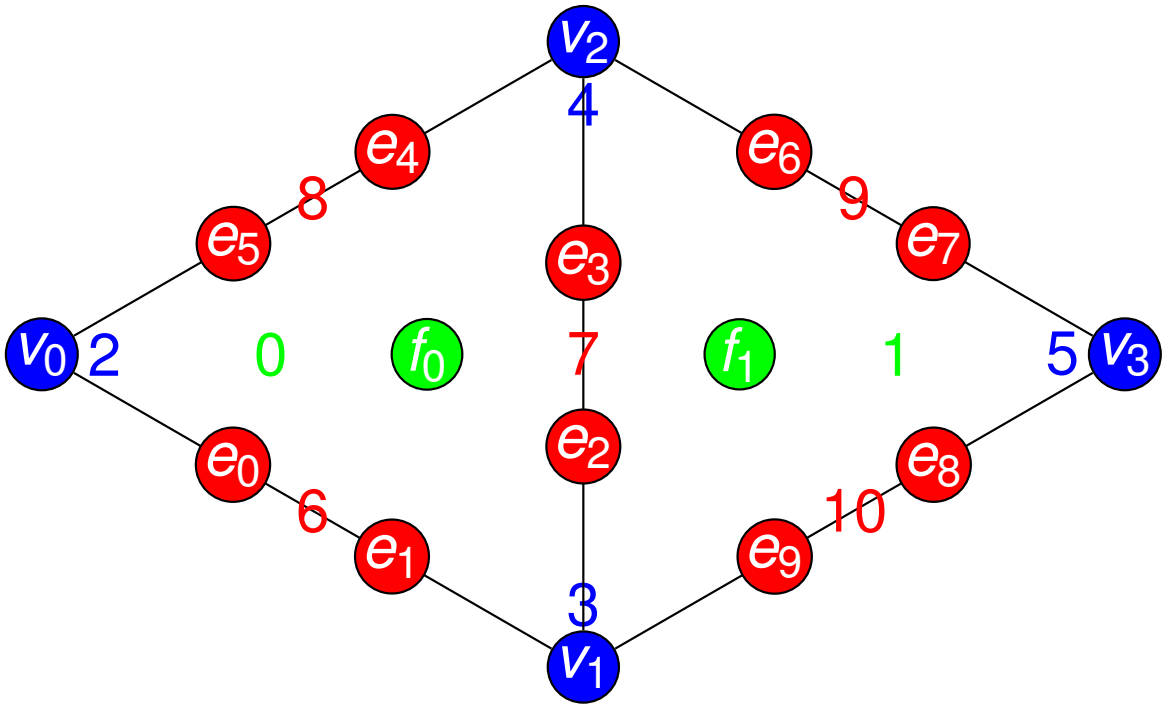
Doublet Section

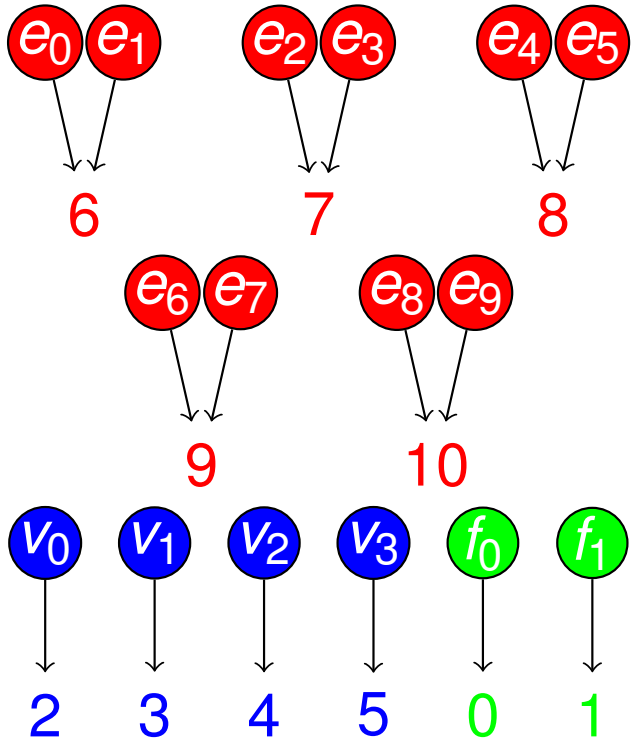




- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

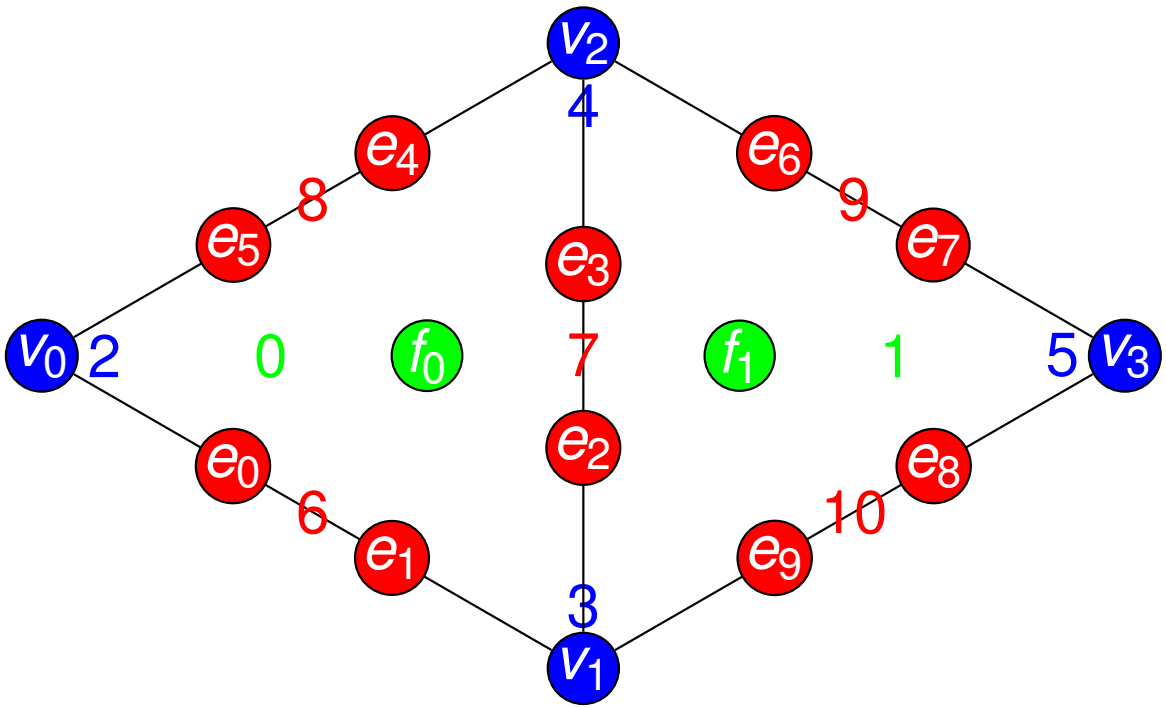
Doublet Section

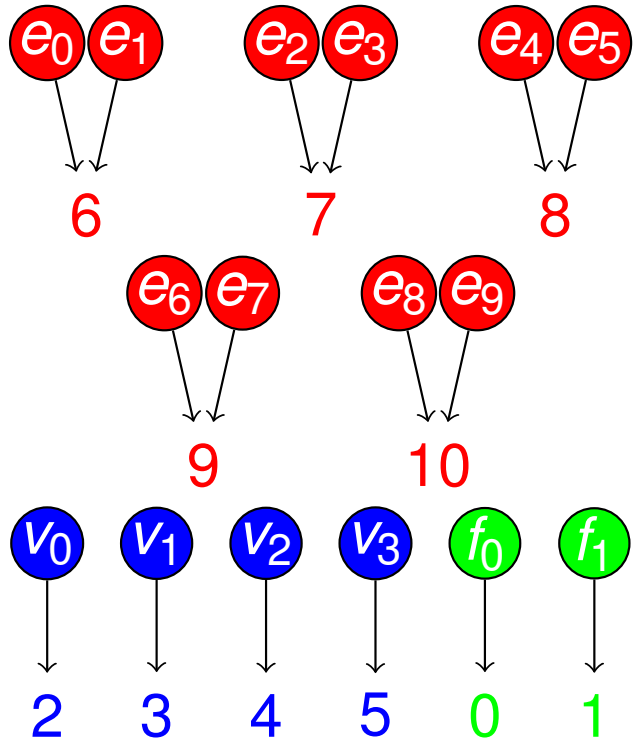




- **Section** interface
  - *restrict*(0) = $\{f_0\}$
  - *restrict*(2) = $\{v_0\}$
  - *restrict*(6) = $\{e_0, e_1\}$

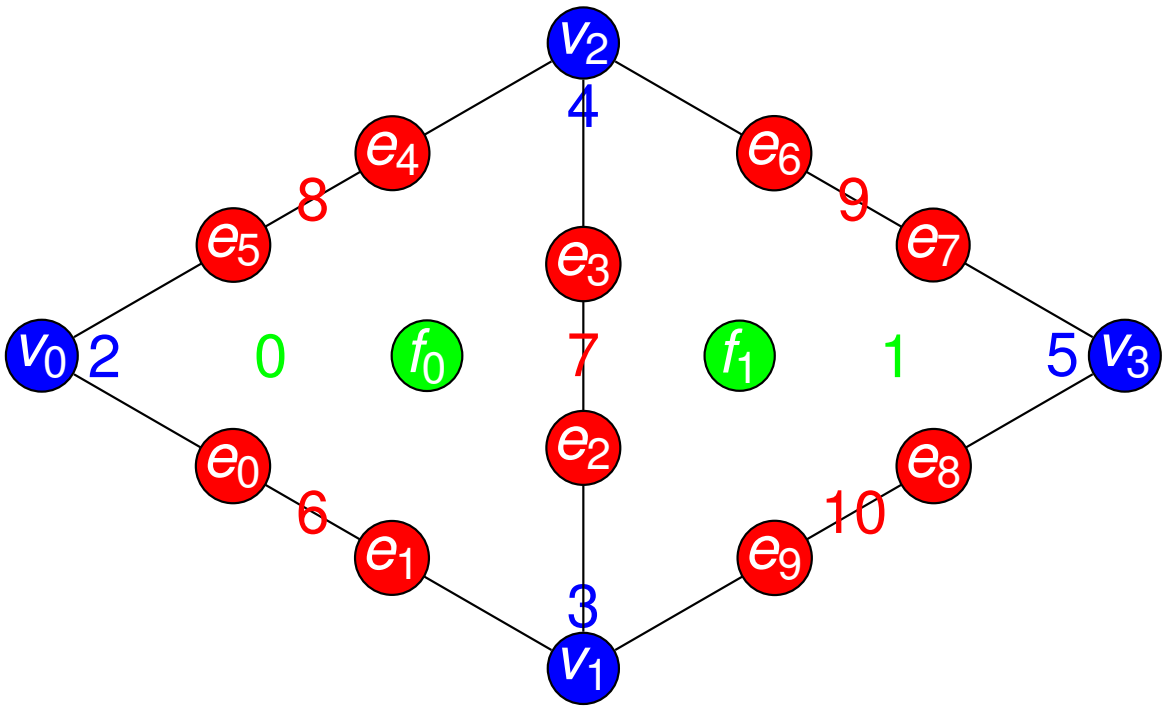
Doublet Section

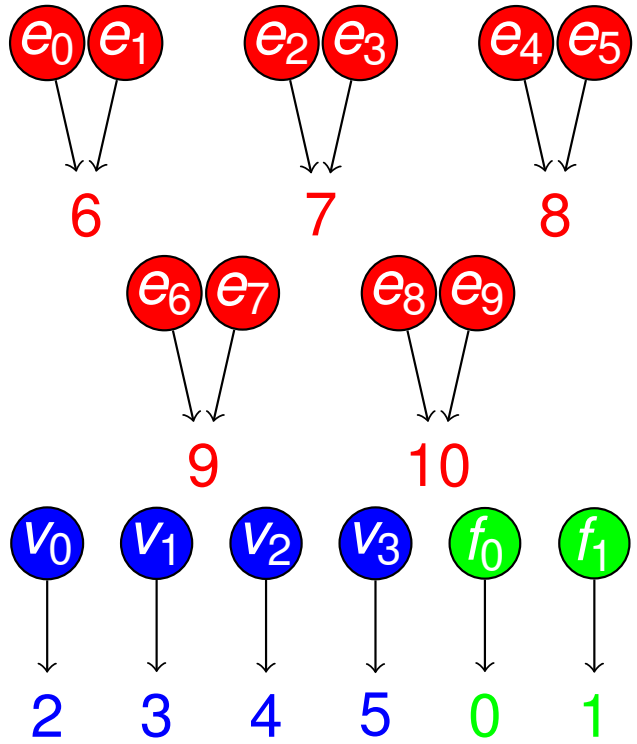




- Topological traversals: follow connectivity
  - $restrictClosure(0) = \{f_0\, e_0\, e_1\, e_2\, e_3\, e_4\, e_5\, v_0\, v_1\, v_2\}$
  - $restrictStar(7) = \{v_0\, e_0\, e_1\, e_4\, e_5\, f_0\}$

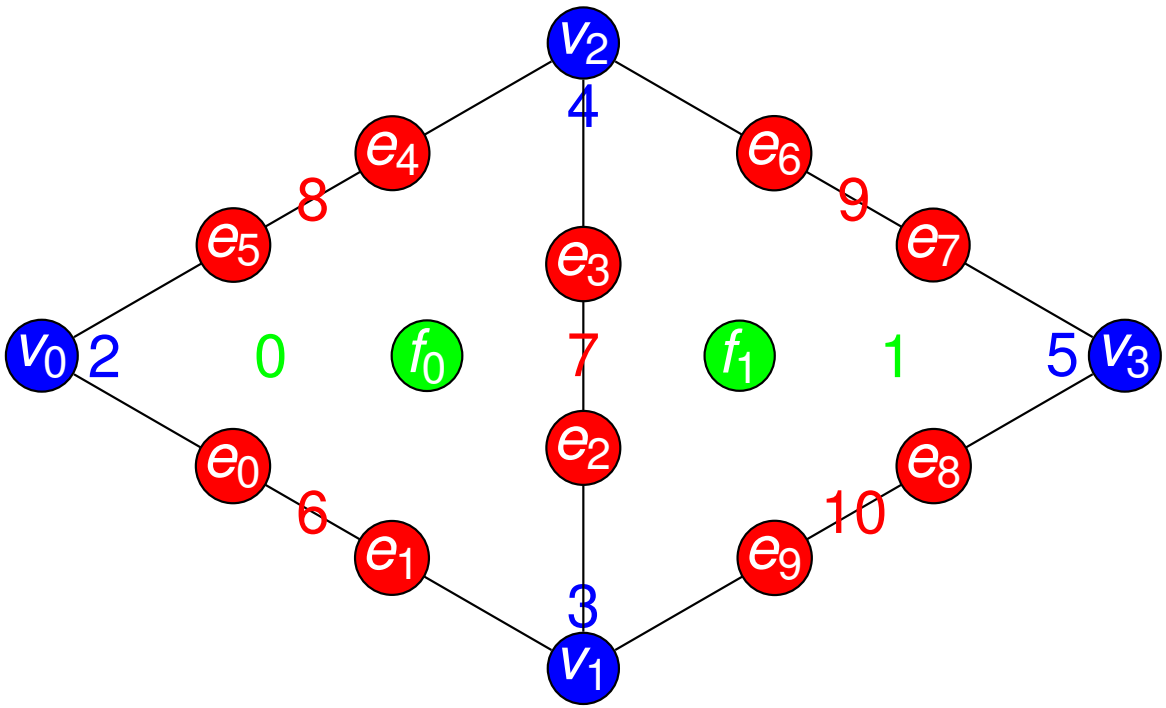
Doublet Section

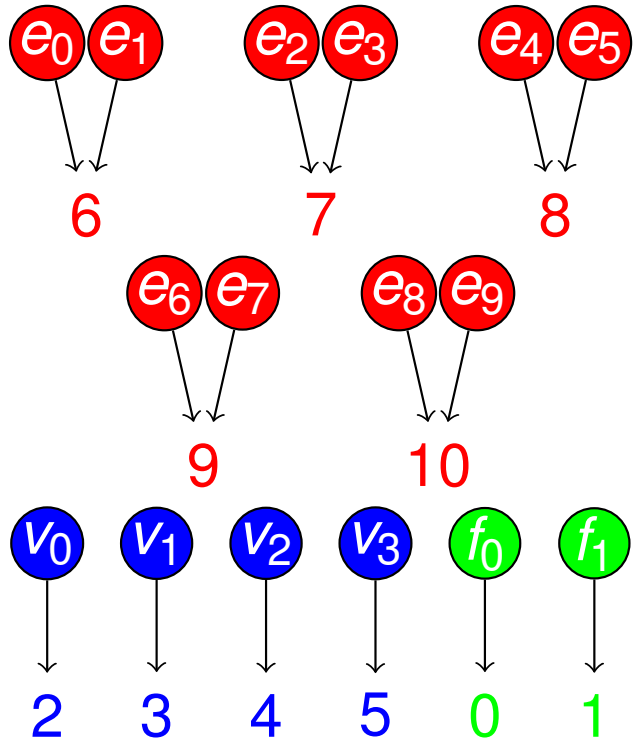




- Topological traversals: follow connectivity
  - $\mathit{restrictClosure}(0) = \{f_0\, e_0\, e_1\, e_2\, e_3\, e_4\, e_5\, v_0\, v_1\, v_2\}$
  - $\mathit{restrictStar}(7) = \{v_0\, e_0\, e_1\, e_4\, e_5\, f_0\}$

Doublet Section





- Topological traversals: follow connectivity
  - $\mathit{restrictClosure}(0) = \{f_0\, e_0\, e_1\, e_2\, e_3\, e_4\, e_5\, v_0\, v_1\, v_2\}$
  - $\mathit{restrictStar}(7) = \{v_0\, e_0\, e_1\, e_4\, e_5\, f_0\}$

Outline

Mapping

Since we have a single relation,

we can see all our objects merely as mappings:

- Section
  - point $\longrightarrow$ real
- Sieve
  - point of S $\longrightarrow$ {points of S}
- Overlap
  - point of S $\longrightarrow$ {points of S'}

Composition

We may compose mappings to generate

- `restrictClosure()`
  - `closure()` $\circ$ `restrictPoint()`
- `updateMeet()`
  - `meet()` $\circ$ `updatePoint()`

and can even compose across an `Overlap`

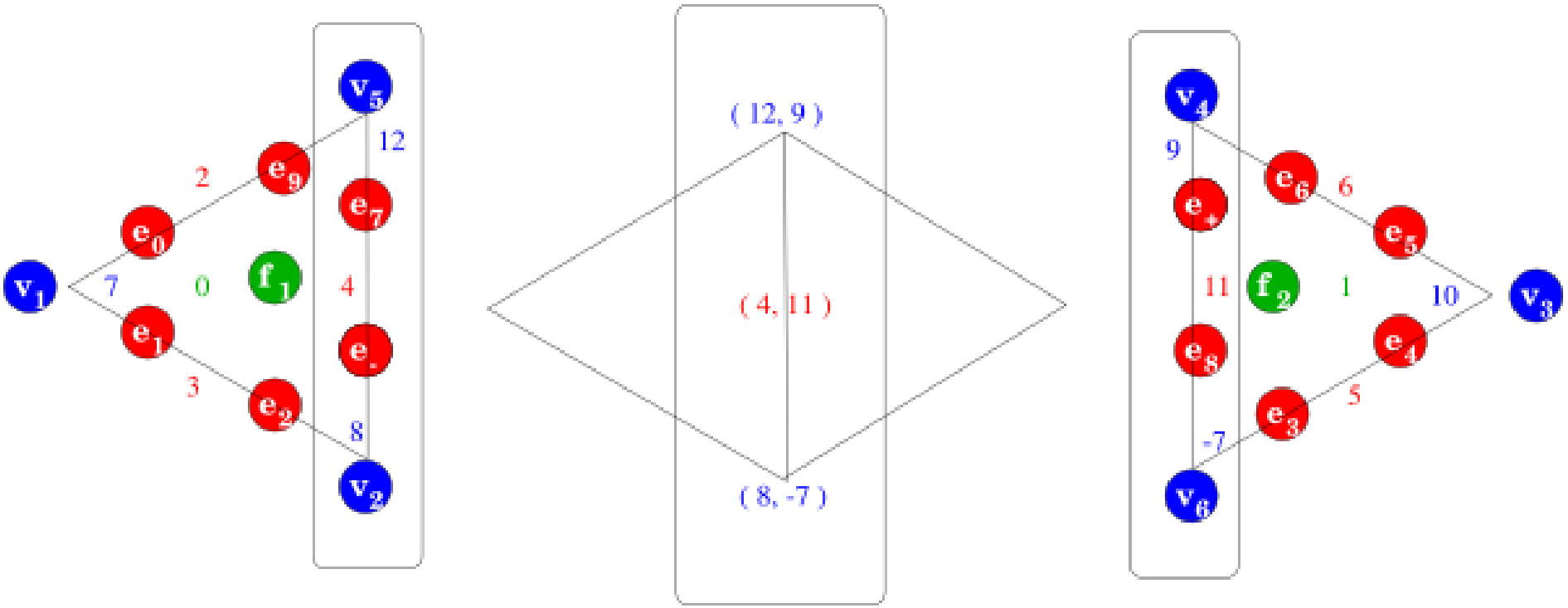
- `complete()` looks like a
  - restriction to the overlap
  - copy between adjacent sieves
  - fusion of values in the overlap sections
  - update to original section

Outline

Sieves of Sieves

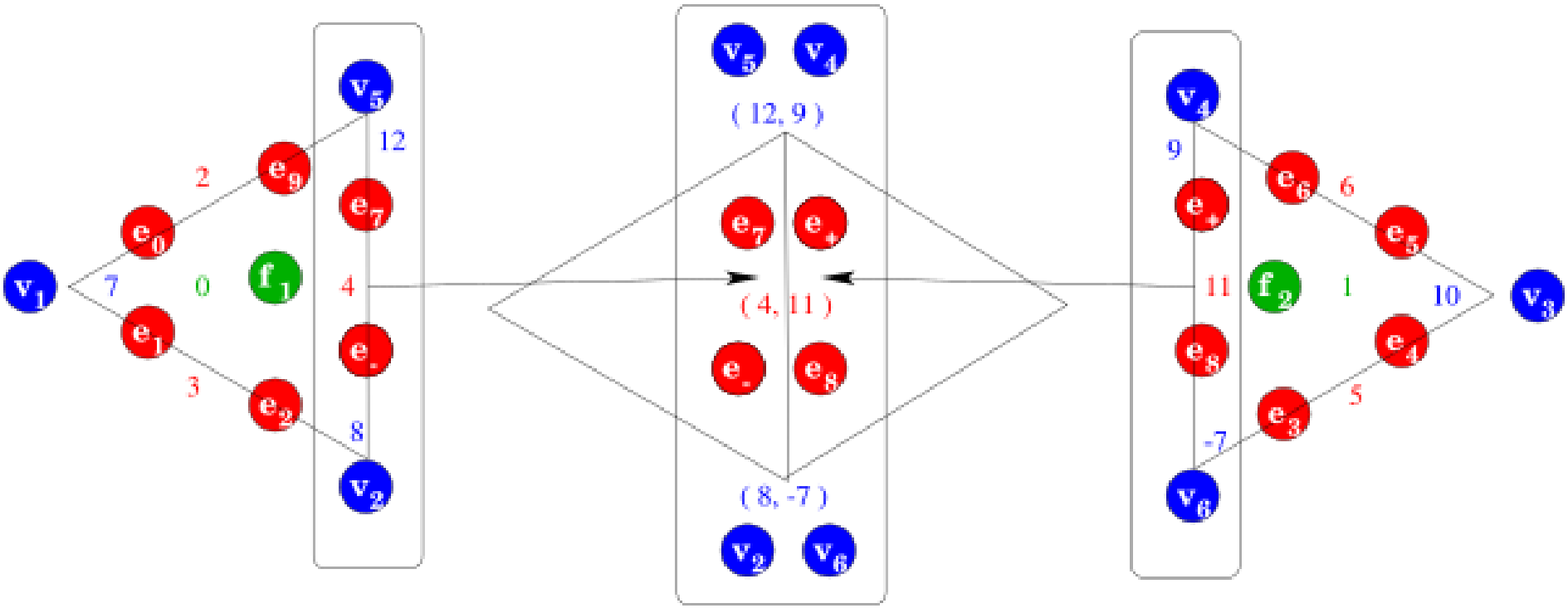
- We can connect two `Sieves` by identifying points
  - This can be seen as nonlocal covering
- This relation is then encapsulated in an `Overlap`,
  - which is just another `Sieve`.
- `Sections` may be defined over the `Overlap`
  - Data movement follows the arrows
- Enforcing consistency across an `Overlap` gives `completion()`

Restriction



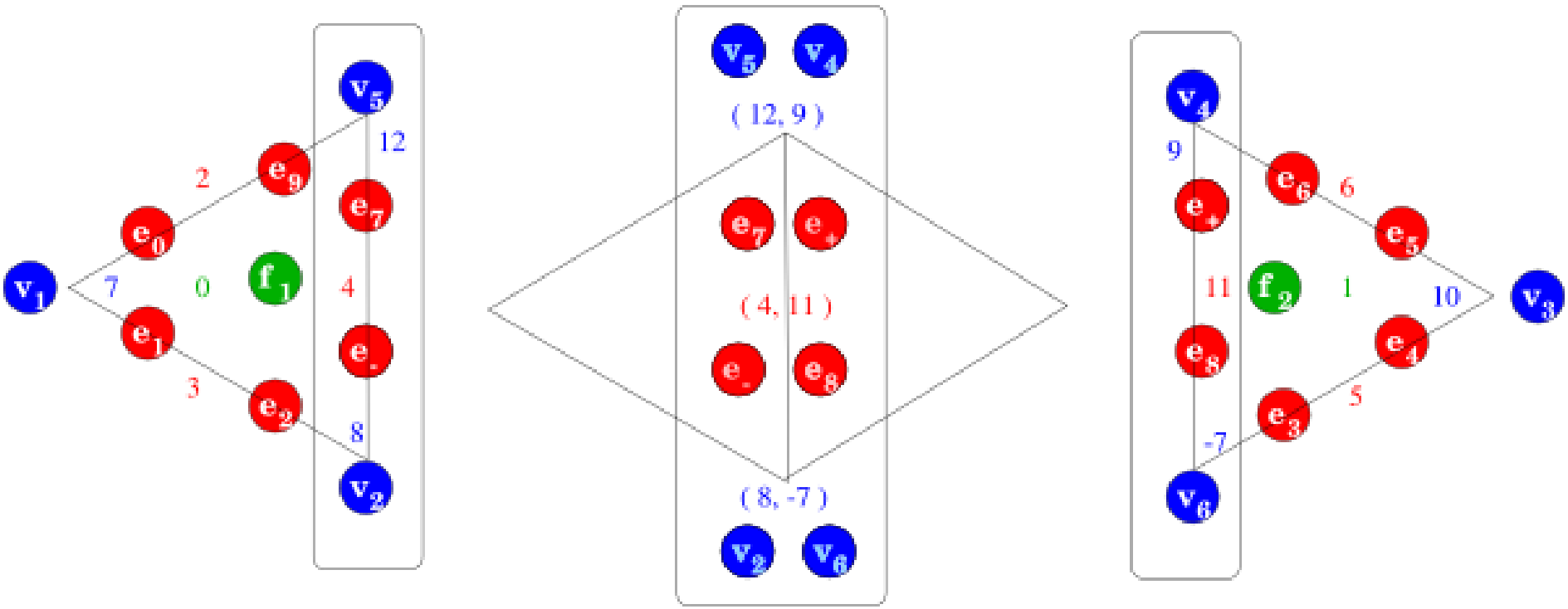
- Localization
  - Restrict to patches (here an edge closure)
  - Compute locally

Delta



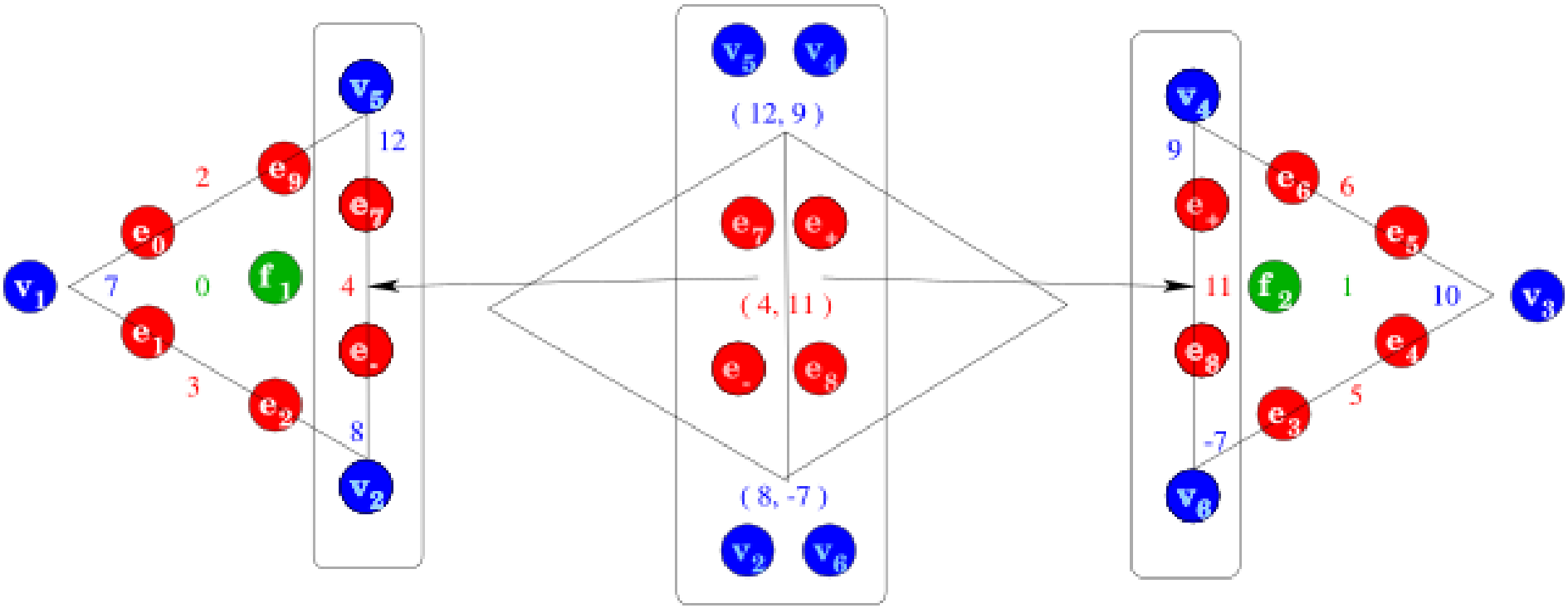
- `Delta`
  - Restrict further to the overlap
  - `Overlap` now carries twice the data

Fusion



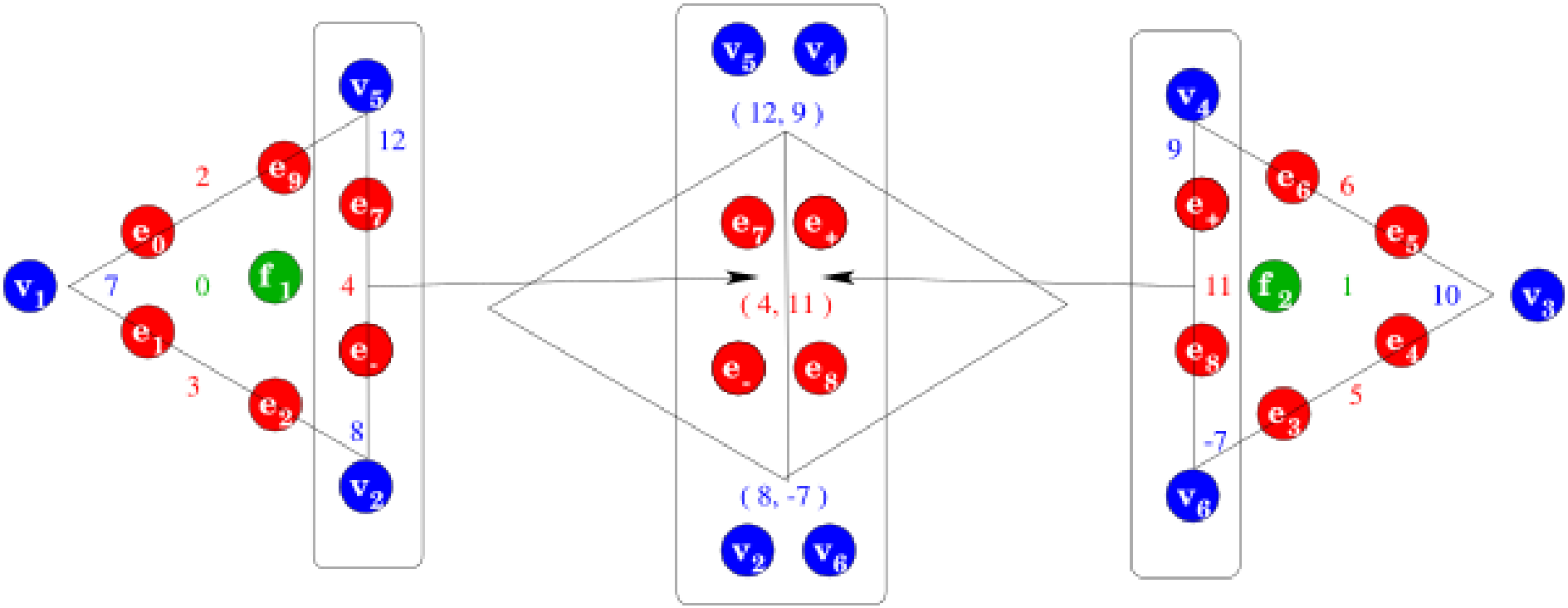
- Merge/reconcile data on the overlap
  - Addition (FEM)
  - Replacement (FD)
  - Coordinate transform (Sphere)
  - Linear transform (MG)

Update



- `Update`
  - Update local patch data
  - `Completion` = restrict $\longrightarrow$ fuse $\longrightarrow$ update, in parallel

Completion



- A ubiquitous parallel form of *restrict* $\longrightarrow$ *fuse* $\longrightarrow$ *update*
- Operates on Sections
  - Sieves can be "downcast" to Sections
- Based on two operations
  - Data exchange through overlap
  - Fusion of shared data

Uses

Completion has many uses:

FEM accumulating integrals on shared faces

FVM accumulating fluxes on shared cells

FDM setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumlating matvec for a partially assembled matrix

Uses

Completion has many uses:

FEM accumulating integrals on shared faces

FVM accumulating fluxes on shared cells

FDM setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumlating matvec for a partially assembled matrix

Uses

Completion has many uses:

FEM accumulating integrals on shared faces

FVM accumulating fluxes on shared cells

FDM setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumlating matvec for a partially assembled matrix

Uses

Completion has many uses:

FEM accumulating integrals on shared faces

FVM accumulating fluxes on shared cells

FDM setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumlating matvec for a partially assembled matrix

Uses

Completion has many uses:

FEM accumulating integrals on shared faces

FVM accumulating fluxes on shared cells

FDM setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumlating matvec for a partially assembled matrix

Uses

Completion has many uses:

FEM accumulating integrals on shared faces

FVM accumulating fluxes on shared cells

FDM setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumlating matvec for a partially assembled matrix

Uses

Completion has many uses:

FEM accumulating integrals on shared faces

FVM accumulating fluxes on shared cells

FDM setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumlating matvec for a partially assembled matrix

Part III

Global Computation: Implementation

Outline

Hierarchical Interfaces

Global/Local Dichotomy is the Heart of DD

Software interfaces do not adequately reflect this

- PETSc DA is too specialized
  - Basically 1D methods applied to Cartesian products
- PETSc Index Sets and VecScatters are too fine
  - User “does everything”, no abstraction
- PETSc Linear Algebra (Vec & Mat) is too coarse
  - No access to the underlying connectivity structure

Unstructured Interface (before)

- Explicit references to element type
  - getVertices(edgeID), getVertices(faceID)
  - getAdjacency(edgeID, VERTEX)
  - getAdjacency(edgeID, dim = 0)
- No interface for transitive closure
  - Awkward nested loops to handle different dimensions
- Have to recode for meshes with different
  - dimension
  - shapes

Unstructured Interface (before)

- Explicit references to element type
  - getVertices(edgeID), getVertices(faceID)
  - getAdjacency(edgeID, VERTEX)
  - getAdjacency(edgeID, dim = 0)
- No interface for transitive closure
  - Awkward nested loops to handle different dimensions
- Have to recode for meshes with different
  - dimension
  - shapes

Unstructured Interface (before)

- Explicit references to element type
  - getVertices(edgeID), getVertices(faceID)
  - getAdjacency(edgeID, VERTEX)
  - getAdjacency(edgeID, dim = 0)
- No interface for transitive closure
  - Awkward nested loops to handle different dimensions
- Have to recode for meshes with different
  - dimension
  - shapes

Go Back to the Math

Combinatorial Topology gives us a framework for geometric computing.

- Abstract to a relation, covering, on sieve points
  - Points can represent any mesh element
  - Covering can be thought of as adjacency
  - Relation can be expressed in a DAG (Hasse Diagram)
- Simple query set:
  - provides a general API for geometric algorithms
  - leads to simpler implementations
  - can be more easily optimized

Go Back to the Math

Combinatorial Topology gives us a framework for geometric computing.

- Abstract to a relation, covering, on sieve points
  - Points can represent any mesh element
  - Covering can be thought of as adjacency
  - Relation can be expressed in a DAG (Hasse Diagram)
- Simple query set:
  - provides a general API for geometric algorithms
  - leads to simpler implementations
  - can be more easily optimized

Go Back to the Math

Combinatorial Topology gives us a framework for geometric computing.

- Abstract to a relation, covering, on sieve points
  - Points can represent any mesh element
  - Covering can be thought of as adjacency
  - Relation can be expressed in a DAG (Hasse Diagram)
- Simple query set:
  - provides a general API for geometric algorithms
  - leads to simpler implementations
  - can be more easily optimized

Unstructured Interface (after)

- NO explicit references to element type
  - A point may be any mesh element
  - getCone(point): adjacent (d-1)-elements
  - getSupport(point): adjacent (d+1)-elements
- Transitive closure
  - closure(cell): The computational unit for FEM
- Algorithms independent of mesh
  - dimension
  - shape (even hybrid)
  - global topology
  - data layout

Unstructured Interface (after)

- NO explicit references to element type
  - A point may be any mesh element
  - getCone(point): adjacent (d-1)-elements
  - getSupport(point): adjacent (d+1)-elements
- Transitive closure
  - closure(cell): The computational unit for FEM
- Algorithms independent of mesh
  - dimension
  - shape (even hybrid)
  - global topology
  - data layout

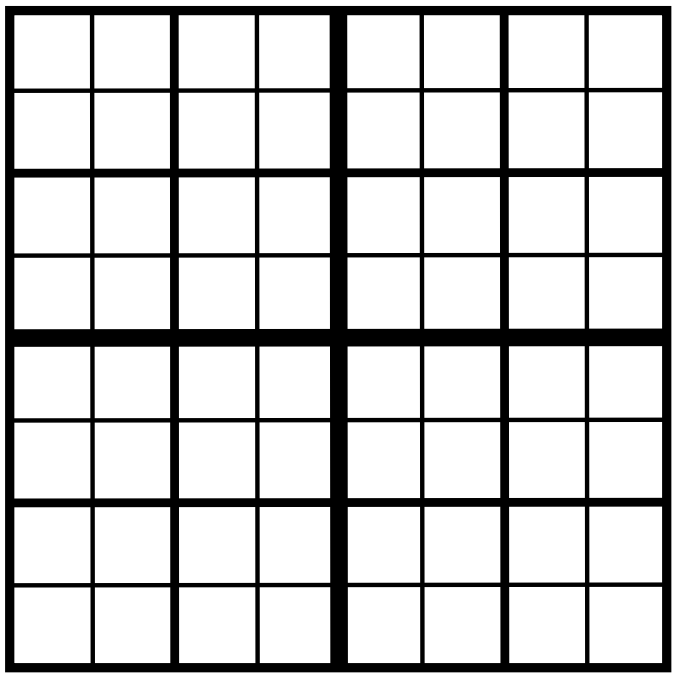
Unstructured Interface (after)

- NO explicit references to element type
  - A point may be any mesh element
  - getCone(point): adjacent (d-1)-elements
  - getSupport(point): adjacent (d+1)-elements
- Transitive closure
  - closure(cell): The computational unit for FEM
- Algorithms independent of mesh
  - dimension
  - shape (even hybrid)
  - global topology
  - data layout

Hierarchy Abstractions

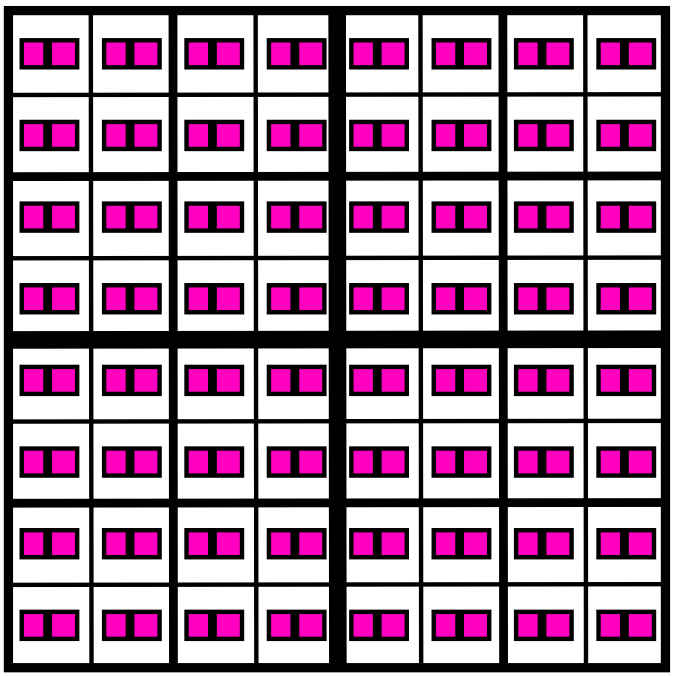
- Generalize to a set of linear spaces
  - `Sieve` provides topology, can also model `Mat`
  - `Section` generalizes `Vec`
  - Spaces interact through an `Overlap` (just a `Sieve`)
- Basic operations
  - Restriction to finer subspaces, `restrict()`/`update()`
  - Assembly to the subdomain, `complete()`
- Allow reuse of geometric and multilevel algorithms

FMM in Sieve



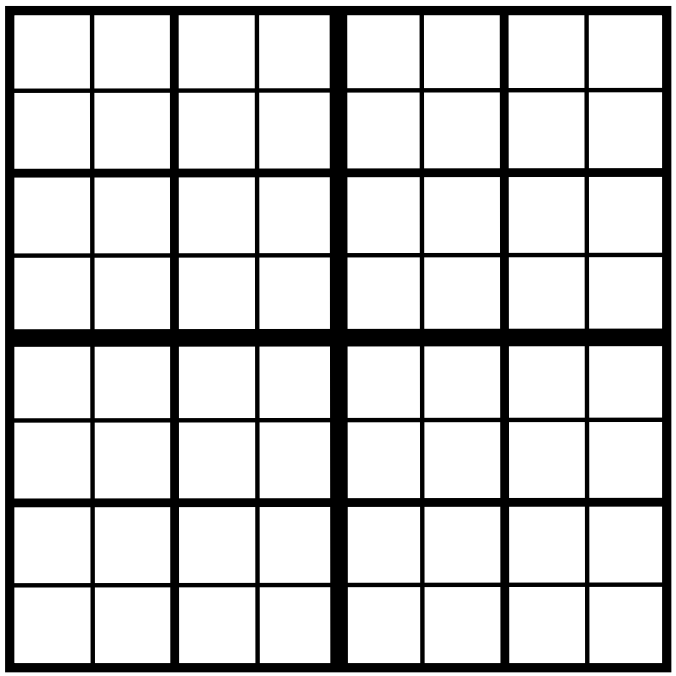
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



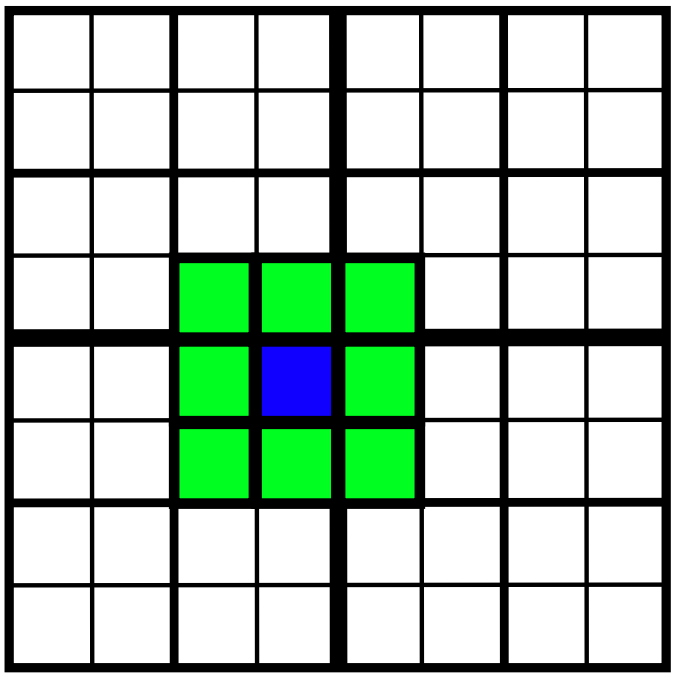
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



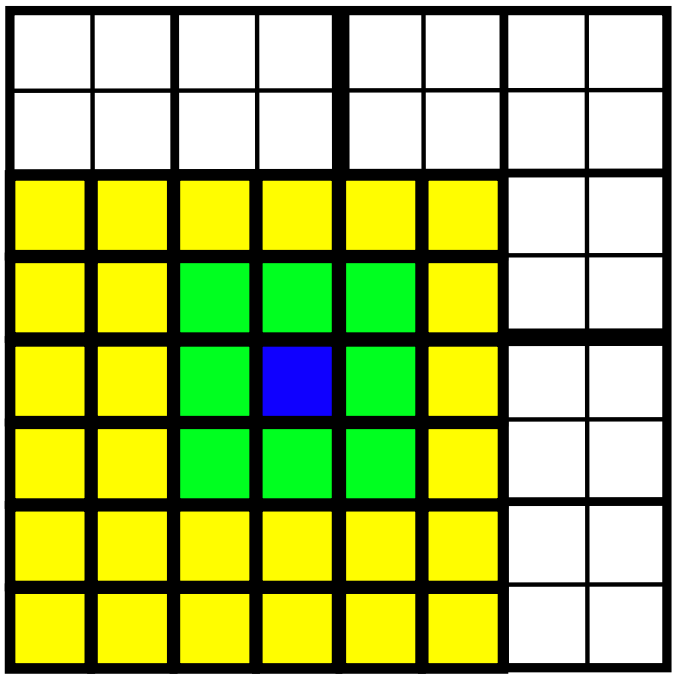
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



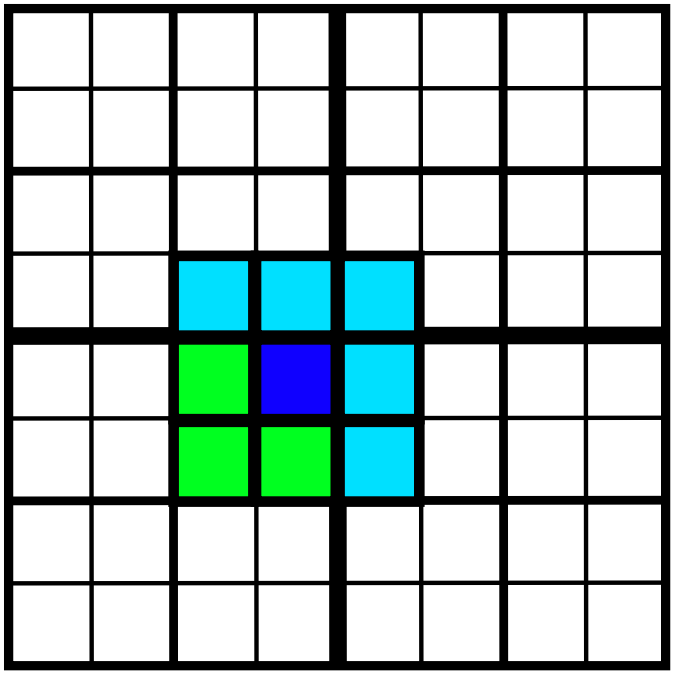
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



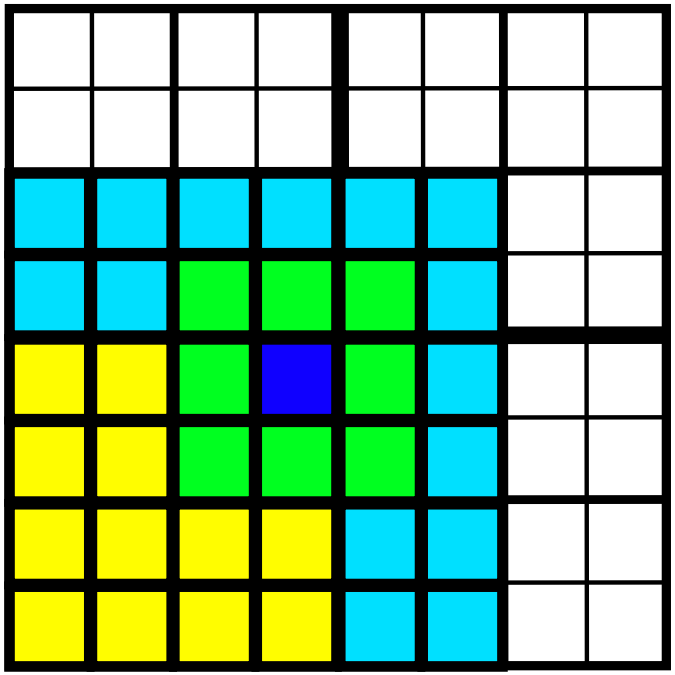
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



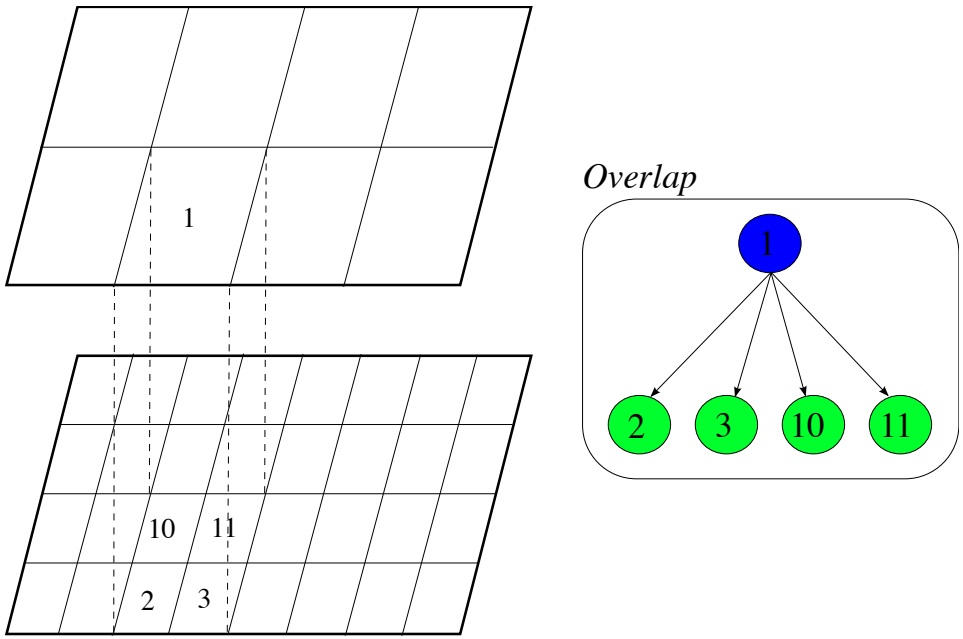
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

Multigrid in Sieve



- `Sieves` represent coarse and fine meshes
- `Sections` represent coarse and fine fields
- An `Overlap` matches coarse and fine cells
- Interpolation and restriction are completion over the overlap
  - Fusion is a linear transformation

Outline

Traversal

Sequences:

- http://en.wikipedia.org/wiki/Iterator_pattern
- State is held by the iterator
- Special classes are unnecessary

```
const sequence& cells = mesh.heightStratum(0);

for(sequence::iterator c_iter = cells.begin();
  c_iter != cells.end(); ++c_iter) {
  point_type p = *c_iter;
}
```

Traversal

Visitors:

- http://en.wikipedia.org/wiki/Visitor_pattern
- State is split between sieve and visitor
- User controls allocation

```
PrintVisitor pV;

sieve.cone(p, pV);
```

Visitor Composition

- Visitors can be composed by chaining `visit()` calls
  - Final template parameter is child visitor type
- `closure()` is accomplished by composition
  - Oriented traversal uses the variant `visit(point, orient)`
- Composition can also proceed by slicing
  - Discussed later by Dmitry

Outline

Section Distribution

Section distribution consists of

- Creation of the local Section
- Distribution of the Atlas (layout Section)
- Completion of the Section

Section Completion

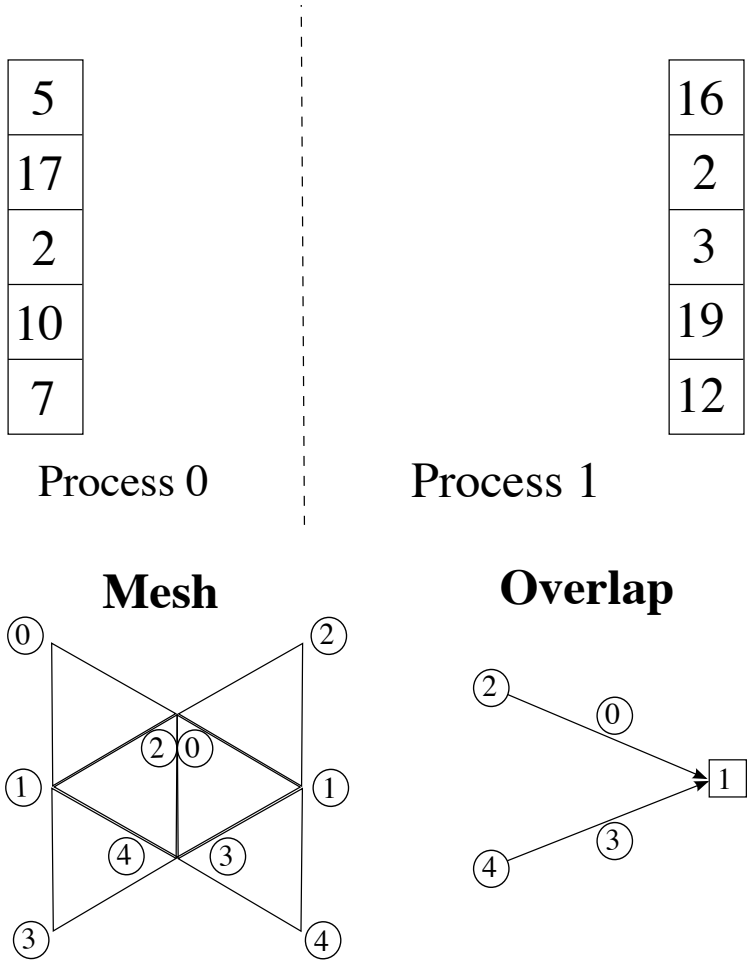
Completion can be broken into 4 phases:

1. `restrict()` to an overlap section
2. `copy()` data to the remote overlap section
3. `fuse()` data with existing point data
4. `update()` remote section with fused overlap section data

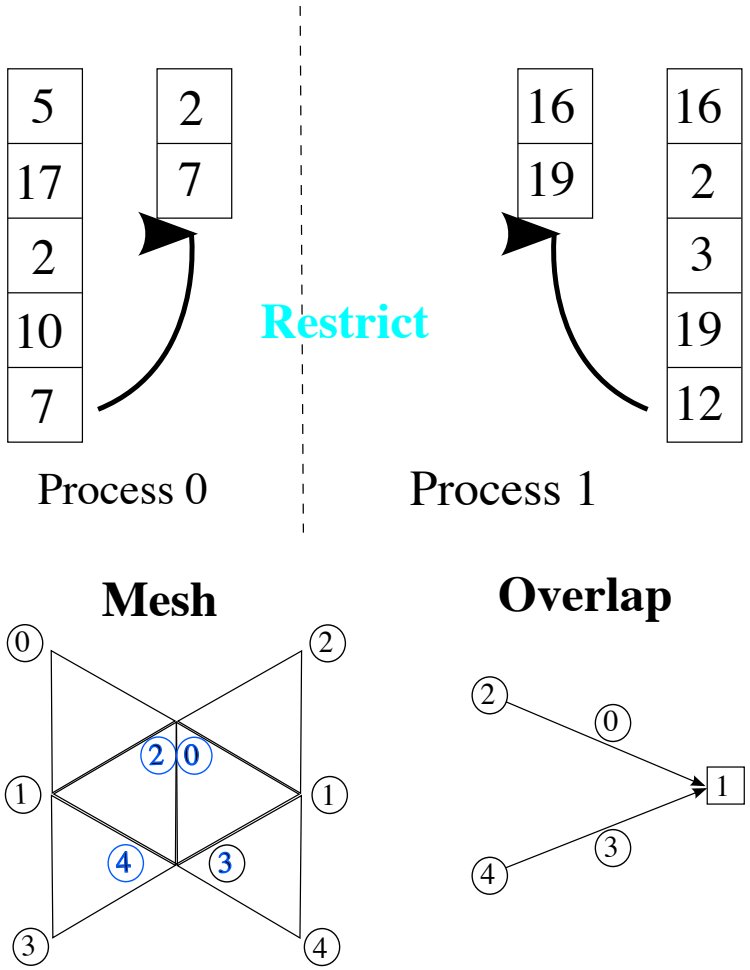
It is common to combine phases 1 & 2, and also 3 & 4

- Data is moved directly between communication buffers and storage

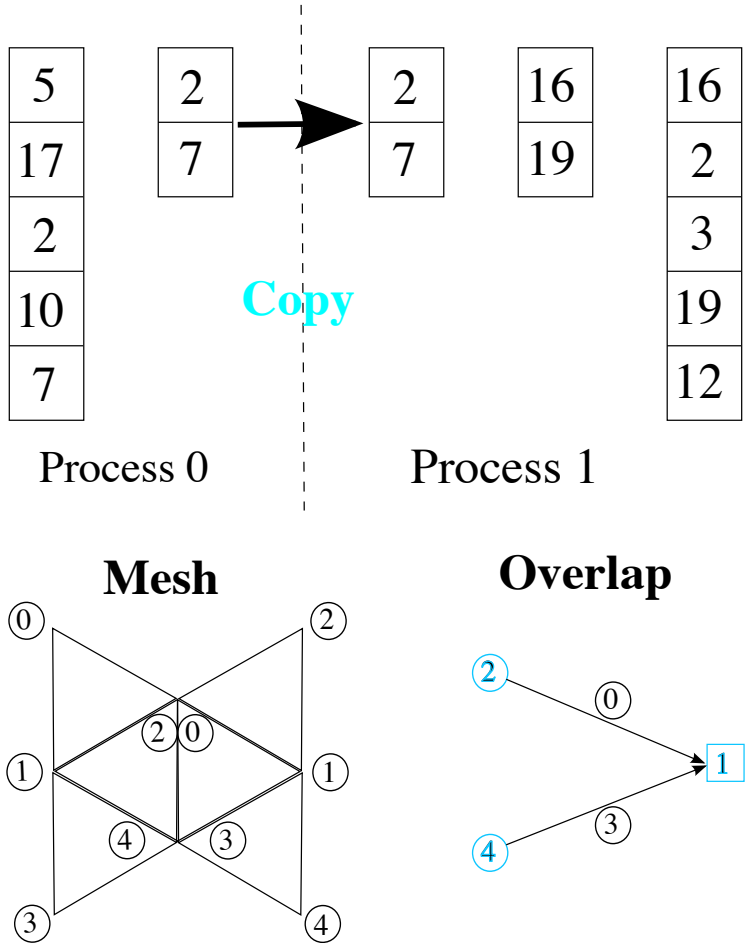
Section Completion



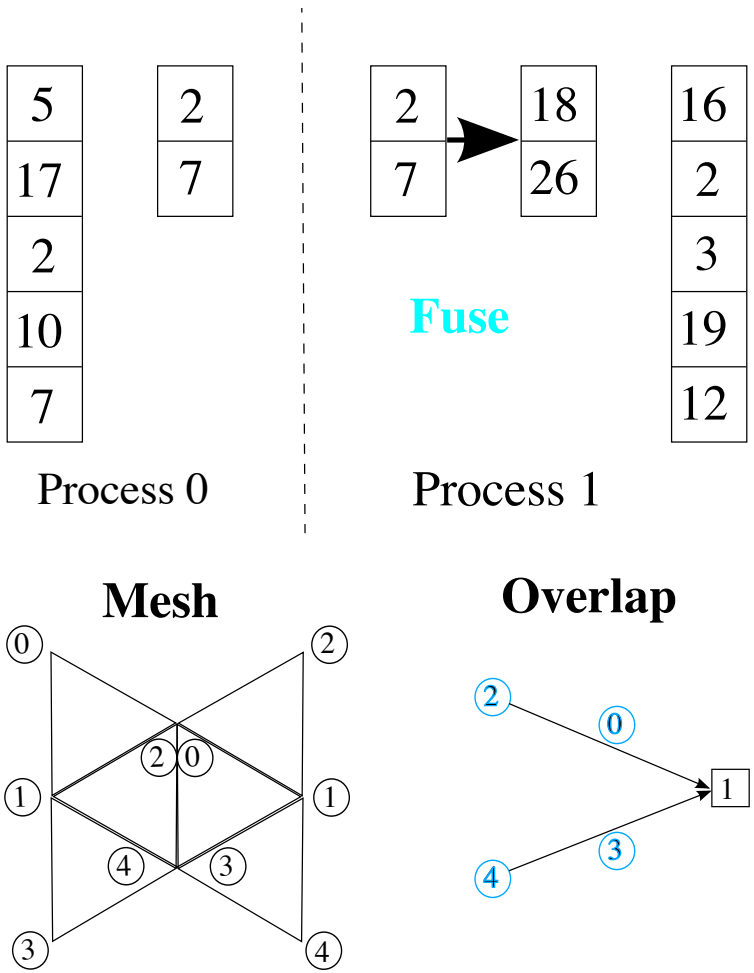
Section Completion



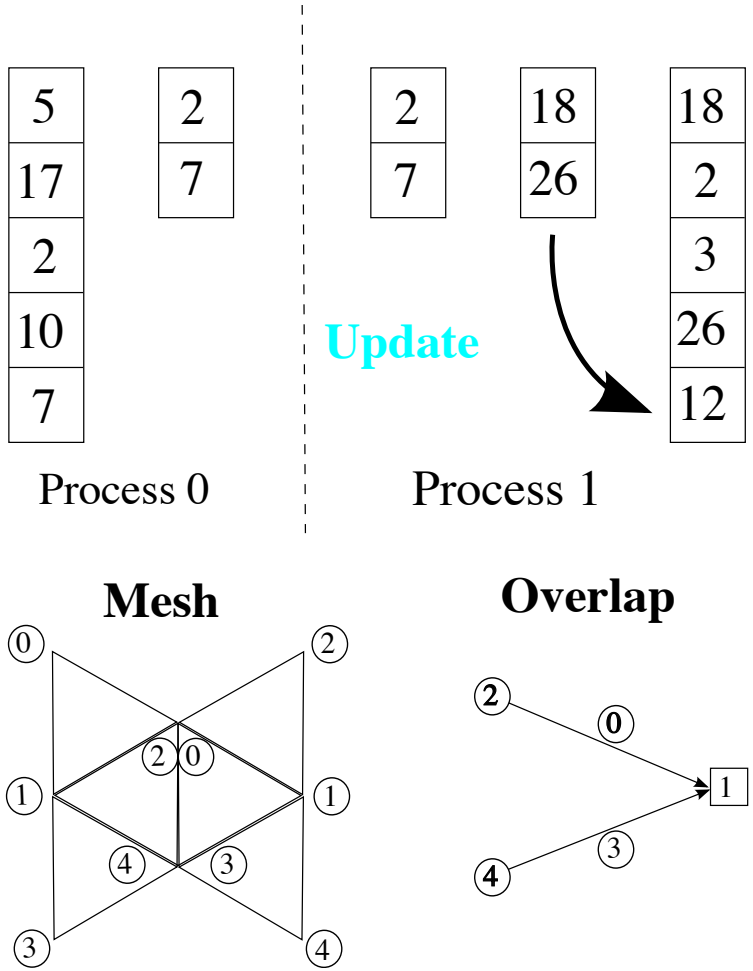
Section Completion



Section Completion



Section Completion



Section Hierarchy

We have a hierarchy of section types of increasing complexity

- `GeneralSection`
  - An arbitrary number of values for each domain point
  - Constrain arbitrary values
  - Atlas is a `UniformSection`
- `UniformSection`
  - A fixed number of values for each domain point
  - Atlas is a `ConstantSection`
- `ConstantSection`
  - The same single value for all domain points
  - Only the domain must be completed

Example: Balanced Matrix-Vector Product

- If a mesh has a highly graded degree sequence,
  - like a power-law (small world) graph,
- `MatMult()` can be very unbalanced
  - since all edges for a vertex must be on one process.
- We can balance edges in local matrices
  - by leaving the partition boundary unassembled.
- We need only `complete()` the output section
  - due to the linearity of the operation

Example: Balanced Matrix-Vector Product

- If a mesh has a highly graded degree sequence,
  - like a power-law (small world) graph,
- `MatMult()` can be very unbalanced
  - since all edges for a vertex must be on one process.
- We can balance edges in local matrices
  - by leaving the partition boundary unassembled.
- We need only `complete()` the output section
  - due to the linearity of the operation

Example: Balanced Matrix-Vector Product

- If a mesh has a highly graded degree sequence,
  - like a power-law (small world) graph,
- `MatMult()` can be very unbalanced
  - since all edges for a vertex must be on one process.
- We can balance edges in local matrices
  - by leaving the partition boundary unassembled.
- We need only `complete()` the output section
  - due to the linearity of the operation

Example: Balanced Matrix-Vector Product

- If a mesh has a highly graded degree sequence,
  - like a power-law (small world) graph,
- `MatMult()` can be very unbalanced
  - since all edges for a vertex must be on one process.
- We can balance edges in local matrices
  - by leaving the partition boundary unassembled.
- We need only `complete()` the output section
  - due to the linearity of the operation

Example: Balanced Matrix-Vector Product

- If a mesh has a highly graded degree sequence,
  - like a power-law (small world) graph,
- `MatMult()` can be very unbalanced
  - since all edges for a vertex must be on one process.
- We can balance edges in local matrices
  - by leaving the partition boundary unassembled.
- We need only `complete()` the output section
  - due to the linearity of the operation

Example: Balanced Matrix-Vector Product

- If a mesh has a highly graded degree sequence,
  - like a power-law (small world) graph,
- `MatMult()` can be very unbalanced
  - since all edges for a vertex must be on one process.
- We can balance edges in local matrices
  - by leaving the partition boundary unassembled.
- We need only `complete()` the output section
  - due to the linearity of the operation

Example: Balanced Matrix-Vector Product

- If a mesh has a highly graded degree sequence,
  - like a power-law (small world) graph,
- `MatMult()` can be very unbalanced
  - since all edges for a vertex must be on one process.
- We can balance edges in local matrices
  - by leaving the partition boundary unassembled.
- We need only `complete()` the output section
  - due to the linearity of the operation

Example: Balanced Matrix-Vector Product

- If a mesh has a highly graded degree sequence,
  - like a power-law (small world) graph,
- `MatMult()` can be very unbalanced
  - since all edges for a vertex must be on one process.
- We can balance edges in local matrices
  - by leaving the partition boundary unassembled.
- We need only `complete()` the output section
  - due to the linearity of the operation

Outline

Outline

Kernels Approach

Reducing operations to kernels is widespread in scientific computing:

- Facilitates code reuse
- Reduces code complexity
- Reduces work of optimization (?)
- Needs correct abstractions

Dual to introducing common software structures

- Kernels operate on common structures

Must enable automatic selection of algorithmic variants

Kernels Approach

Reducing operations to kernels is widespread in scientific computing:

- Facilitates code reuse
- Reduces code complexity
- Reduces work of optimization (?)
- Needs correct abstractions

Dual to introducing common software structures

- Kernels operate on common structures

Must enable automatic selection of algorithmic variants

Kernels Approach

Reducing operations to kernels is widespread in scientific computing:

- Facilitates code reuse
- Reduces code complexity
- Reduces work of optimization (?)
- Needs correct abstractions

Dual to introducing common software structures

- Kernels operate on common structures

Must enable automatic selection of algorithmic variants

Dense Linear Algebra

Dense linear algebra is too rich:

- Rich structure allow many different organizations
- BLAS/LAPACK chooses certain kernel operations
  - Consider only reuse, not optimization
- LAPACK choose a single variant of each algorithm
- LAPACK fixes the structure implementation in the interface

FLAME allows new kernels to be created

- Abstracts among implementations (layouts)

DFT

Spiral allows both reuse and optimization:

- Abstract model from algorithms
- Allows different implementations for common structures
- Automates algorithm selection
- Incorporates performance feedback

Unfortunately, DFT is simpler than our common operations.

DFT

Spiral allows both reuse and optimization:

- Abstract model from algorithms
- Allows different implementations for common structures
- Automates algorithm selection
- Incorporates performance feedback

Unfortunately, DFT is simpler than our common operations.

Sparse Linear Algebra

Sparse linear algebra has a single kernel (SpMV):

- Don't specify our algorithms at the FLAME level
  - Without a PME, cannot move between variants automatically
- Can be built from Sieve completion operations
  - Completion of operator gives assembled matrix
  - Completion of output gives matrix-free application
- `VecScatter` should be generalized to an `Overlap`

Sparse Linear Algebra

Sparse linear algebra has a single kernel (SpMV):

- Don't specify our algorithms at the FLAME level
  - Without a PME, cannot move between variants automatically
- Can be built from Sieve completion operations
  - Completion of operator gives assembled matrix
  - Completion of output gives matrix-free application
- `VecScatter` should be generalized to an `Overlap`

Sparse Linear Algebra

Sparse linear algebra has a single kernel (SpMV):

- Don't specify our algorithms at the FLAME level
  - Without a PME, cannot move between variants automatically
- Can be built from Sieve completion operations
  - Completion of operator gives assembled matrix
  - Completion of output gives matrix-free application
- `VecScatter` should be generalized to an `Overlap`

Sparse Linear Algebra

Sparse linear algebra has a single kernel (SpMV):

- Don't specify our algorithms at the FLAME level
  - Without a PME, cannot move between variants automatically
- Can be built from Sieve completion operations
  - Completion of operator gives assembled matrix
  - Completion of output gives matrix-free application
- `VecScatter` should be generalized to an `Overlap`

Performance Insights

There are two key insights for automatic performance tuning:

1. Memory layout controls performance (Goto)
   - Must be able to switch layouts for different algorithmic variants
     - Bad LAPACK interface truncates ATLAS search space
   - Example: GEPP kernel for DGEMM
2. Must understand data dependencies
   - OpenMP cannot express this
   - Can be encapsulated in a DAG
     - SuperMatrix
     - Sieve
   - Enables variants switching (loop fusion)

Performance Insights

There are two key insights for automatic performance tuning:

1. Memory layout controls performance (Goto)
   - Must be able to switch layouts for different algorithmic variants
     - Bad LAPACK interface truncates ATLAS search space
   - Example: GEPP kernel for DGEMM
2. Must understand data dependencies
   - OpenMP cannot express this
   - Can be encapsulated in a DAG
     - SuperMatrix
     - Sieve
   - Enables variants switching (loop fusion)

Sieve and Overlap

Sieve and Overlap can structure computation by expression of

- Hierarchy
  - Reduces complexity and enables code reuse with
    - common components (sieve)
    - operations (completion)
  - Separates global and local concerns
  - Maps well to multiresolution algorithms
- Dependency
  - Allows tranformation between different algorithmic variants
  - Applies at many levels
    - algorithm selection
    - serial scheduling
    - parallel coordination
  - Key advance over Map-Reduce paradigm

Sieve and Overlap

Sieve and Overlap can structure computation by expression of

- Hierarchy
  - Reduces complexity and enables code reuse with
    - common components (sieve)
    - operations (completion)
  - Separates global and local concerns
  - Maps well to multiresolution algorithms
- Dependency
  - Allows tranformation between different algorithmic variants
  - Applies at many levels
    - algorithm selection
    - serial scheduling
    - parallel coordination
  - Key advance over Map-Reduce paradigm

Outline

MPICH-G2

Early Attempt at Hierarchy

- Communicator hierarchy, topology depth
- Only exposed to the user through Comm attributes
  - Still have to support flat model
- Hierarchy information is buried too deep
  - Only really accessible in the implementation (collectives)

MPICH-G2

Early Attempt at Hierarchy

- Communicator hierarchy, topology depth
- Only exposed to the user through Comm attributes
  - Still have to support flat model
- Hierarchy information is buried too deep
  - Only really accessible in the implementation (collectives)

MPICH-G2

Early Attempt at Hierarchy

- Communicator hierarchy, topology depth
- Only exposed to the user through Comm attributes
  - Still have to support flat model
- Hierarchy information is buried too deep
  - Only really accessible in the implementation (collectives)

Hierarchy in MPI

MPI communicator should be imbued with hierarchy:

- Single relation is easy to add
  - Could be implemented using attributes
- Can easily code hierarchical algorithms
  - FMM, MG, ...
- Can express data dependencies
  - Communicator could represent a thread group
  - Scheduling could be done inside MPI interface (SuperMatrix)
- Enables large and small scale parallelism
  - Domain decomposition
  - Master-slave
- Could be proposed in MPI-3

Outline

Completion Optimization

A `Section` with unchanged structure need not recomplete its Atlas

- The `Overlap` could store the packing information and buffers
  - A `VecScatter` could be created between buffers
- For simple fusers, the `Overlap` maps directly to section storage
  - A `VecScatter` could be created between the arrays

Outline

Interval Sieves and Sections

We can demand that our chart be an interval:

- Membership is $\mathcal{O}(1)$
- `cone()` is $\mathcal{O}(1)$
- `restrict()` is $\mathcal{O}(1)$

Formerly, all point queries were $\mathcal{O}(\log n)$

Moreover, no storage is needed for a search structure:

- STL sets require 20 bytes/int

We can always achieve this in a static setting with local renumbering

ISieve

`ISieve`

- Separate AIJ structures for cones and supports
- Also store AIJ orientations
- Must call `allocate()` before setting cones
- Some support for dynamic insertion
- Cones and supports unconnected
  - Use `symmetrize()` to automate arrow reversal
- Has converter from standard `Sieve`
- Visitors for all traversals

ISection

`ISection`

- AIJ structure for values
- Same `allocate()` call before setting values
- Some support for dynamic insertion
- Completion must still send chart explicitly
  - Can amortize across similar completions

Outline

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}
<Aggregate updates>
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  coords = mesh->restrict(coordinates, c);
  v0, J, invJ, detJ = computeGeometry(coords);
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}
<Aggregate updates>
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  inputVec = mesh->restrict(U, c);
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}
<Aggregate updates>
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}
<Aggregate updates>
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    realCoords = J*refCoords[q] + v0;
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}
<Aggregate updates>
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}
<Aggregate updates>
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      elemVec[f] += basis[q,f]*rhsFunc(realCoords);
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}
<Aggregate updates>
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}
<Aggregate updates>
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      for(d = 0; d < dim; ++d)
        for(e) testDerReal[d] += invJ[e,d]*basisDer[q,
      for(g = 0; g < numBasisFuncs; ++g) {
        for(d = 0; d < dim; ++d)
          for(e) basisDerReal[d] += invJ[e,d]*basisDer
          elemMat[f,g] += testDerReal[d]*basisDerReal[
        elemVec[f] += elemMat[f,g]*inputVec[g];
      }
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}
<Aggregate updates>
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      elemVec[f] += basis[q,f]*lambda*exp(inputVec[f])
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}
<Aggregate updates>
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}
<Aggregate updates>
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q]*detJ;
    }
  }
  mesh->updateAdd(F, c, elemVec);
}
<Aggregate updates>
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}
<Aggregate updates>
```

Integration

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}
Distribution<Mesh>::completeSection(mesh, F);
```

Outline

Boundary Conditions

Dirichlet conditions may be expressed as

$$u|_\Gamma = g$$

and implemented by constraints on dofs in a Section

- The user provides a function.

Neumann conditions may be expressed as

$$\nabla u \cdot \hat{n}|_\Gamma = h$$

and implemented by explicit integration along the boundary

- The user provides a weak form.

Dual Basis Application

We would like the action of a dual basis vector (functional)

$$< \mathcal{N}_i, f >= \int_{\text{ref}} N_i(x) f(x) dV$$

- Projection onto $\mathcal{P}$
- Code is generated from FIAT specification
  - Python code generation package inside PETSc
- Common interface for all elements

Assembly with Dirichlet Conditions

The original equation may be partitioned into

- unknowns in the interior (I)
- unknowns on the boundary (Γ)

so that we obtain

$$\begin{pmatrix} A_{II} & A_{I\Gamma} \\ A_{\Gamma I} & A_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} u_I \\ u_\Gamma \end{pmatrix} = \begin{pmatrix} f_I \\ f_\Gamma \end{pmatrix}$$

However u_Γ is known, so we may reduce this to

$$A_{II} u_I = f_I - A_{I\Gamma} u_\Gamma$$

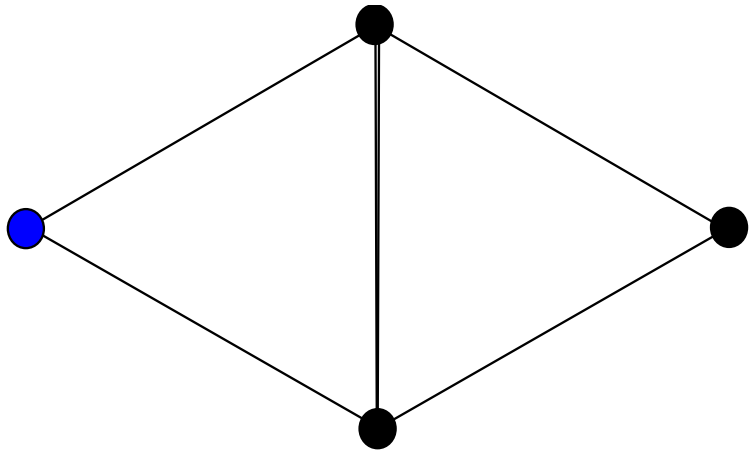
We will show that our scheme automatically constructs this extra term.

Assembly with Dirichlet Conditions

Residual Assembly

u	**5**	**1**	**3**	**7**

f	**5**	**0**	**0**	**0**

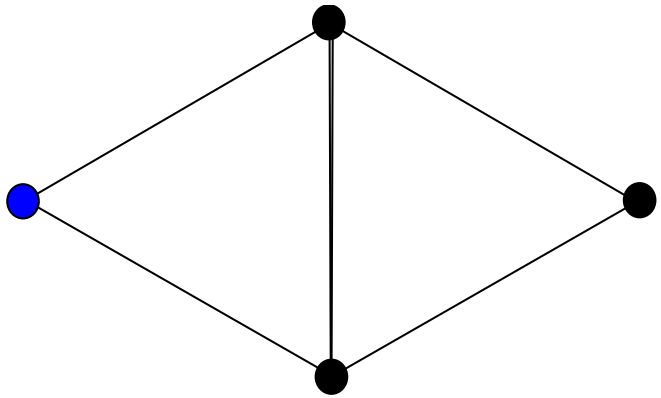


Assembly with Dirichlet Conditions

Residual Assembly

u				
	5	1	3	7

f				
	5	0	0	0



Restrict

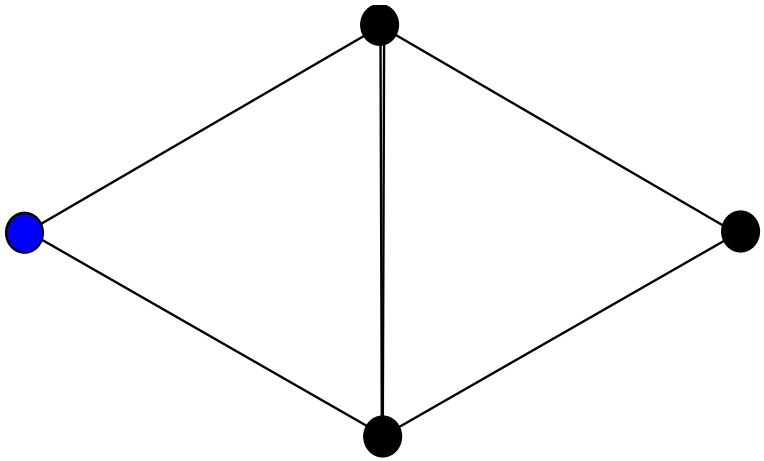
5
1
3

Assembly with Dirichlet Conditions

Residual Assembly

u				
	5	1	3	7

f				
	5	0	0	0



Compute

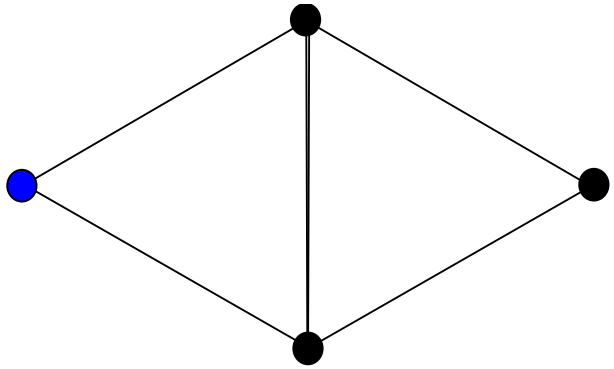
$$\begin{bmatrix} 0.5 & 0.0 & -0.5 \\ 0.0 & 0.5 & -0.5 \\ -0.5 & -0.5 & 1.0 \end{bmatrix} \begin{bmatrix} 5 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$$

Assembly with Dirichlet Conditions

Residual Assembly

u				
	5	1	3	7

f				
	5	0	0	0



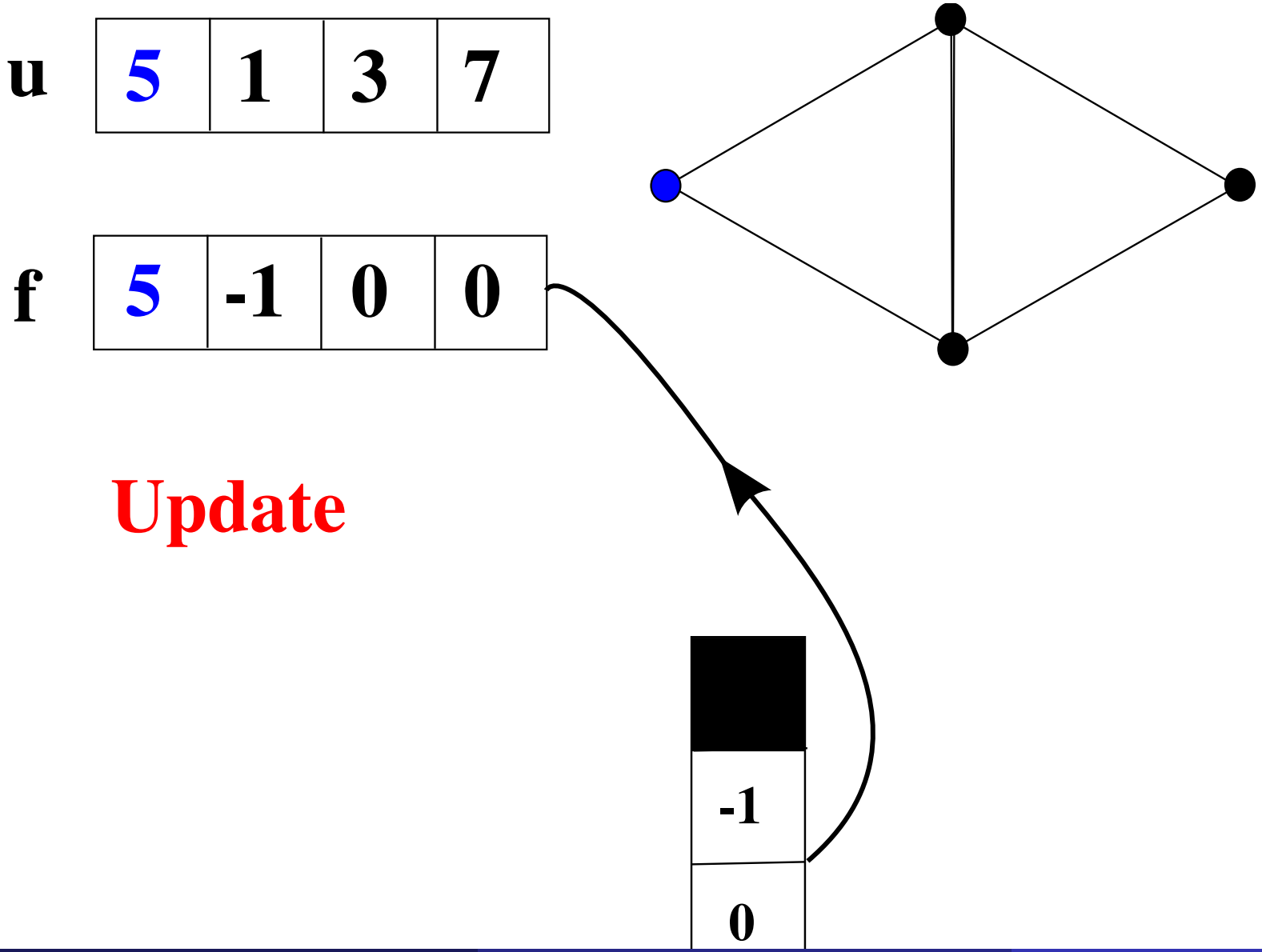
Compute

$$\begin{bmatrix} A_{\Gamma\Gamma} & A_{\Gamma I} \\ A_{I\Gamma} & A_{II} \end{bmatrix} \begin{bmatrix} 5 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$$

This piece containsrhs

Assembly with Dirichlet Conditions

Residual Assembly

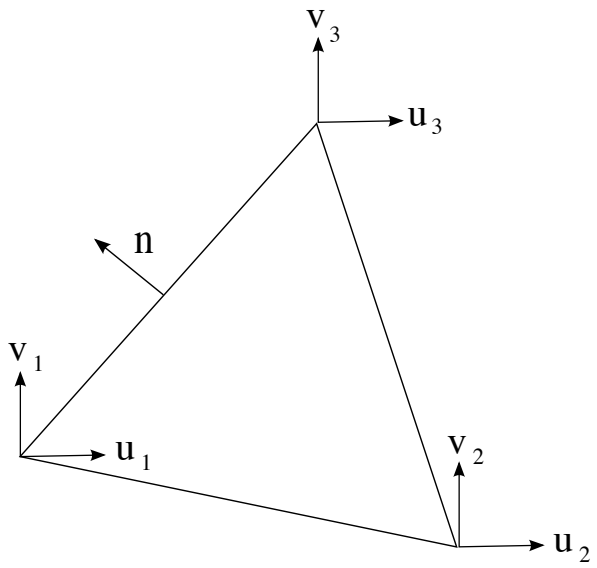


Dirichlet Values

- Topological boundary is marked during generation
- Cells bordering boundary are marked using `markBoundaryCells()`
- To set values:
  1. Loop over boundary cells
  2. Loop over the element closure
  3. For each boundary point i, apply the functional N_i to the function g
- The functionals are generated with the quadrature information
- Section allocation applies Dirichlet conditions automatically
  - Values are stored in the Section
  - `restrict()` behaves normally, `update()` ignores constraints

Complex BC

We may want to constrain a dof not in the global basis:



For instance, no flow normal to a boundary

$$\hat{n} \cdot \mathbf{v} = 0$$

when the global basis follows the coordinate directions.

Complex BC

- In order to constrain the value we
  - rotate the storage coordinates to the $n - \tau$ frame
  - project out the normal coordinate (freeze the value)
- This rotation is also needed for restriction
  - and any action accessing section storage
- In general, we need
  - a transformation to BC coordinates
  - a projection onto free variables (trivial)
- Transformation might involve all element variables
  - which would be an action on the closure

Part IV

Local Computation: Theory

Outline

Outline

Form Decomposition

Element integrals are decomposed into analytic and geometric parts:

$$\int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) d\mathbf{x} \tag{1}$$

$$= \int_{\mathcal{T}} \frac{\partial\phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial\phi_j(\mathbf{x})}{\partial x_\alpha} d\mathbf{x} \tag{2}$$

$$= \int_{\mathcal{T}_{\text{ref}}} \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\xi_\gamma}{\partial x_\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} |J| d\mathbf{x} \tag{3}$$

$$= \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} d\mathbf{x} \tag{4}$$

$$= G^{\beta\gamma}(\mathcal{T}) K^{ij}_{\beta\gamma} \tag{5}$$

Coefficients are also put into the geometric part.

Form Decomposition

Additional fields give rise to multilinear forms.

$$\int_{\mathcal{T}} \phi_i(\mathbf{x}) \cdot \left(\phi_k(\mathbf{x}) \nabla \phi_j(\mathbf{x})\right) dA \tag{6}$$

$$= \int_{\mathcal{T}} \phi_i^\beta(\mathbf{x}) \left(\phi_k^\alpha(\mathbf{x}) \frac{\partial \phi_j^\beta(\mathbf{x})}{\partial x_\alpha}\right) dA \tag{7}$$

$$= \int_{\mathcal{T}_{\text{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} |J| dA \tag{8}$$

$$= \frac{\partial \xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} dA \tag{9}$$

$$= G^{\alpha\gamma}(\mathcal{T}) K_{\alpha\gamma}^{ijk} \tag{10}$$

The index calculus is fully developed by Kirby and Logg in A Compiler for Variational Forms.

Form Decomposition

Isoparametric Jacobians also give rise to multilinear forms

$$\int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) dA \tag{11}$$

$$= \int_{\mathcal{T}} \frac{\partial\phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial\phi_j(\mathbf{x})}{\partial x_\alpha} dA \tag{12}$$

$$= \int_{\mathcal{T}_{\text{ref}}} \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\xi_\gamma}{\partial x_\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} |J| dA \tag{13}$$

$$= |J| \int_{\mathcal{T}_{\text{ref}}} \phi_k J_k^{\beta\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \phi_l J_l^{\gamma\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} dA \tag{14}$$

$$= J_k^{\beta\alpha} J_l^{\gamma\alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \phi_k \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \phi_l \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} dA \tag{15}$$

$$= G_{kl}^{\beta\gamma}(\mathcal{T}) K_{\beta\gamma}^{ijkl} \tag{16}$$

A different space could also be used for Jacobians

Outline

Outline

Assembly with Dirichlet Conditions

The original equation may be partitioned into

- unknowns in the interior (I)
- unknowns on the boundary (Γ)

so that we obtain

$$\begin{pmatrix} A_{II} & A_{I\Gamma} \\ A_{\Gamma I} & A_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} u_I \\ u_\Gamma \end{pmatrix} = \begin{pmatrix} f_I \\ f_\Gamma \end{pmatrix}$$

However u_Γ is known, so we may reduce this to

$$A_{II} u_I = f_I - A_{I\Gamma} u_\Gamma$$

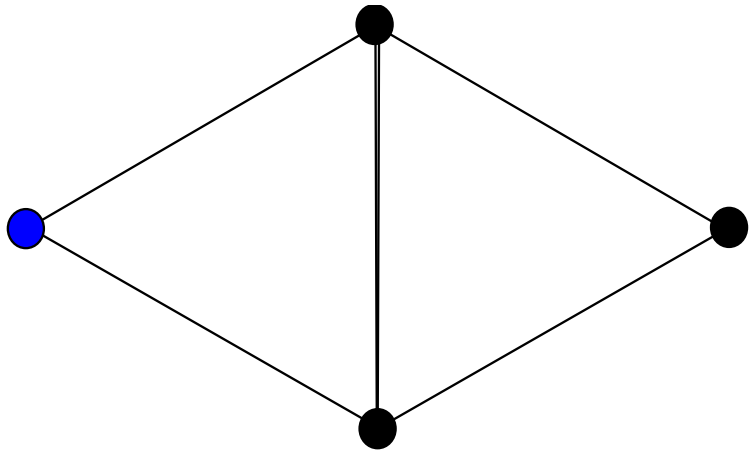
We will show that our scheme automatically constructs this extra term.

Assembly with Dirichlet Conditions

Residual Assembly

u	**5**	**1**	**3**	**7**

f	**5**	**0**	**0**	**0**



Assembly with Dirichlet Conditions

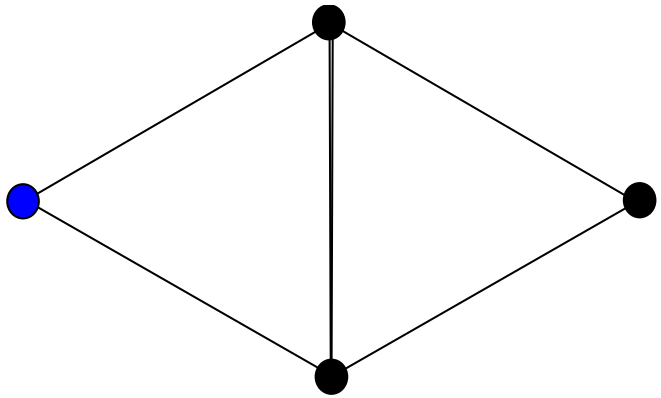
Residual Assembly

u				
	5	1	3	7

f				
	5	0	0	0

Restrict

5
1
3

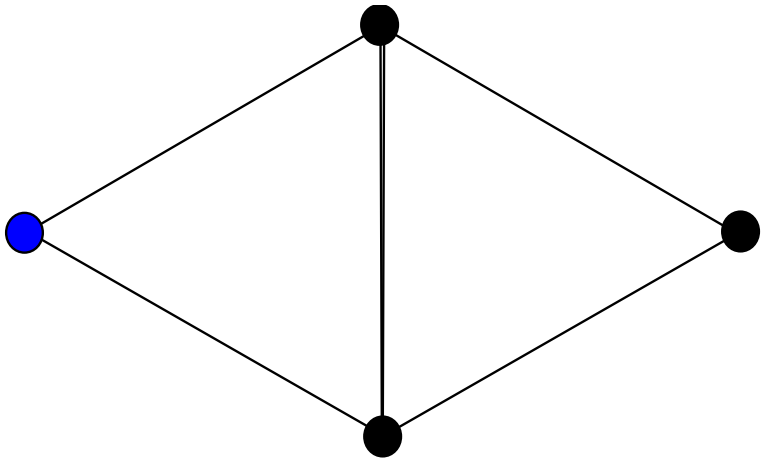


Assembly with Dirichlet Conditions

Residual Assembly

u				
	5	1	3	7

f				
	5	0	0	0



Compute

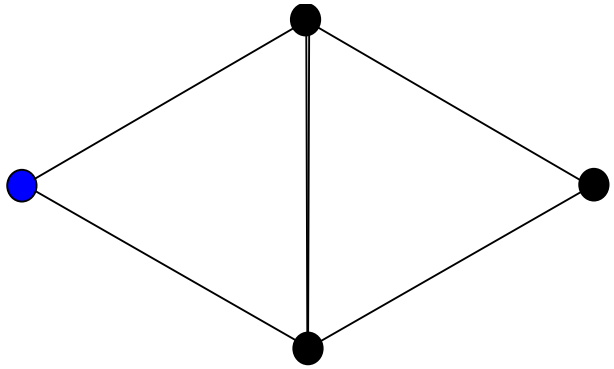
$$\begin{bmatrix} 0.5 & 0.0 & -0.5 \\ 0.0 & 0.5 & -0.5 \\ -0.5 & -0.5 & 1.0 \end{bmatrix} \begin{bmatrix} 5 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$$

Assembly with Dirichlet Conditions

Residual Assembly

u				
	5	1	3	7

f				
	5	0	0	0



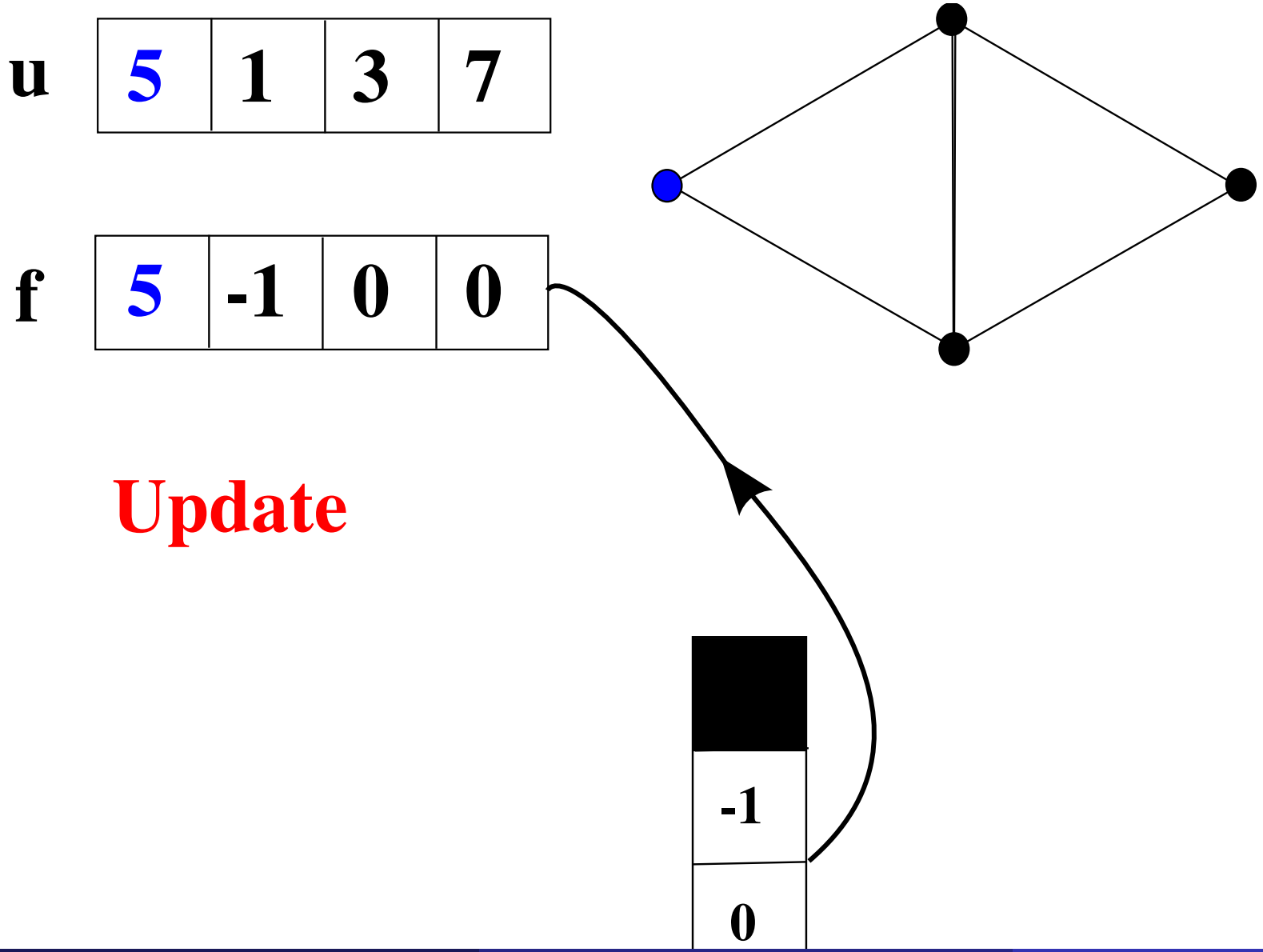
Compute

$$\begin{bmatrix} A_{\Gamma\Gamma} & A_{\Gamma I} \\ A_{I\Gamma} & A_{II} \end{bmatrix} \begin{bmatrix} 5 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$$

} This piece containsrhs

Assembly with Dirichlet Conditions

Residual Assembly

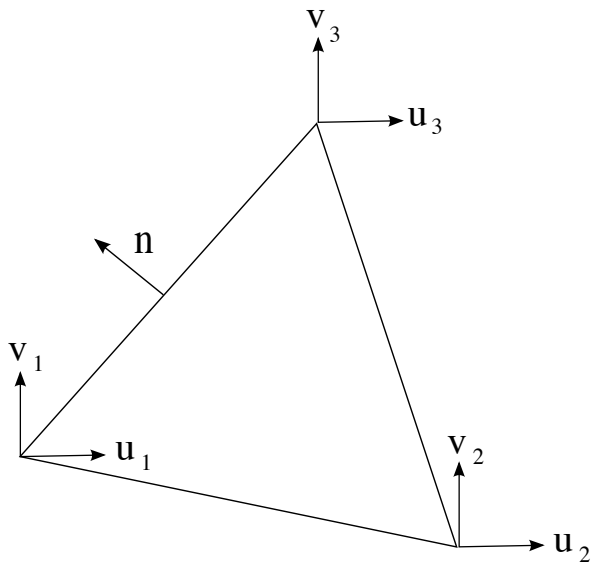


Dirichlet Values

- Topological boundary is marked during generation
- Cells bordering boundary are marked using `markBoundaryCells()`
- To set values:
  1. Loop over boundary cells
  2. Loop over the element closure
  3. For each boundary point i, apply the functional N_i to the function g
- The functionals are generated with the quadrature information
- Section allocation applies Dirichlet conditions automatically
  - Values are stored in the Section
  - `restrict()` behaves normally, `update()` ignores constraints

Complex BC

We may want to constrain a dof not in the global basis:



For instance, no flow normal to a boundary

$$\hat{n} \cdot \mathbf{v} = 0$$

when the global basis follows the coordinate directions.

Complex BC

- In order to constrain the value we
  - rotate the storage coordinates to the $n - \tau$ frame
  - project out the normal coordinate (freeze the value)
- This rotation is also needed for restriction
  - and any action accessing section storage
- In general, we need
  - a transformation to BC coordinates
  - a projection onto free variables (trivial)
- Transformation might involve all element variables
  - which would be an action on the closure

Outline

FFC

FFC is a compiler for variational forms by Anders Logg.

Here is a mixed-form Poisson equation:

$$a((\tau, w), (\sigma, u)) = L((\tau, w)) \qquad \forall (\tau, w) \in V$$

where

$$\begin{aligned} a((\tau, w), (\sigma, u)) &= \int_\Omega \tau\sigma - \nabla \cdot \tau u + w \nabla \cdot u \, dx \\ L((\tau, w)) &= \int_\Omega wf \, dx \end{aligned}$$

FFC
Mixed Poisson

```
shape = "triangle"

BDM1 = FiniteElement("Brezzi-Douglas-Marini",shape,1)
DG0  = FiniteElement("Discontinuous Lagrange",shape,0)

element     = BDM1 + DG0
(tau, w)    = TestFunctions(element)
(sigma, u)  = TrialFunctions(element)

a = (dot(tau, sigma) - div(tau)*u + w*div(sigma))*dx

f = Function(DG0)
L = w*f*dx
```

FFC

Here is a discontinuous Galerkin formulation of the Poisson equation:

$$a(v,u) = L(v) \qquad \forall v \in V$$

where

$$\begin{aligned}
a(v,u) &= \int_\Omega \nabla u \cdot \nabla v \, dx \\
&+ \sum_S \int_S - < \nabla v > \cdot [[u]]_n - [[v]]_n \cdot < \nabla u > - (\alpha/h) vu \, dS \\
&+ \int_{\partial\Omega} -\nabla v \cdot [[u]]_n - [[v]]_n \cdot \nabla u - (\gamma/h) vu \, ds \\
L(v) &= \int_\Omega vf \, dx
\end{aligned}$$

FFC
DG Poisson

```
DG1 = FiniteElement("Discontinuous Lagrange",shape,1)
v = TestFunctions(DG1)
u = TrialFunctions(DG1)
f = Function(DG1)
g = Function(DG1)
n = FacetNormal("triangle")
h = MeshSize("triangle")
a = dot(grad(v), grad(u))*dx
  - dot(avg(grad(v)), jump(u, n))*dS
  - dot(jump(v, n), avg(grad(u)))*dS
  + alpha/h*dot(jump(v, n) + jump(u, n))*dS
  - dot(grad(v), jump(u, n))*ds
  - dot(jump(v, n), grad(u))*ds
  + gamma/h*v*u*ds
L = v*f*dx + v*g*ds
```

Part V

Local Computation: Implementation

Outline

STREAM Benchmark

Simple benchmark program measuring sustainable memory bandwidth

- Protoypical operation is Triad (WAXPY): $\mathbf{w} = \mathbf{y} + \alpha\mathbf{x}$
- Measures the memory bandwidth bottleneck (much below peak)
- Datasets outstrip cache

Machine	Peak (MF/s)	Triad (MB/s)	MF/MW	Eq. MF/s
Matt's Laptop	1700	1122.4	12.1	93.5 (5.5%)
Intel Core2 Quad	38400	5312.0	57.8	442.7 (1.2%)
Tesla 1060C	984000	102000.0*	77.2	8500.0 (0.8%)

Table: Bandwidth limited machine performance

http://www.cs.virginia.edu/stream/

Analysis of Sparse Matvec (SpMV)

Assumptions

- No cache misses
- No waits on memory references

Notation

m	Number of matrix rows
nz	Number of nonzero matrix elements
V	Number of vectors to multiply

We can look at bandwidth needed for peak performance

$$\left(8 + \frac{2}{V}\right)\frac{m}{nz} + \frac{6}{V} \text{ byte/flop} \tag{17}$$

or achieveable performance given a bandwith BW

$$\frac{Vnz}{(8V+2)m + 6nz} BW \text{ Mflop/s} \tag{18}$$

Towards Realistic Performance Bounds for Implicit CFD Codes, Gropp, Kaushik, Keyes, and Smith.

Improving Serial Performance

For a single matvec with 3D FD Poisson, Matt's laptop can achieve at most

$$\frac{1}{(8+2)\frac{1}{7}+6}\ \text{bytes/flop}(1122.4\ \text{MB/s}) = 151\ \text{MFlops/s}, \tag{19}$$

which is a dismal 8.8% of peak.

Can improve performance by

- Blocking
- Multiple vectors

but operation issue limitations take over.

Improving Serial Performance

For a single matvec with 3D FD Poisson, Matt's laptop can achieve at most

$$\frac{1}{(8+2)\frac{1}{7}+6}\ \text{bytes/flop}(1122.4\ \text{MB/s}) = 151\ \text{MFlops/s}, \tag{19}$$

which is a dismal 8.8% of peak.

Better approaches:

- Unassembled operator application (Spectral elements, FMM)
  - N data, N^2 computation
- Nonlinear evaluation (Picard, FAS, Exact Polynomial Solvers)
  - N data, N^k computation

Performance Tradeoffs

We must balance storage, bandwidth, and cycles

- Assembled Operator Action
  - Trades cycles and storage for bandwidth in application
- Unassembled Operator Action
  - Trades bandwidth and storage for cycles in application
  - For high orders, storage is impossible
  - Can make use of FErari decomposition to save calculation
  - Could storage element matrices to save cycles
- Partial assembly gives even finer control over tradeoffs
  - Also allows introduction of parallel costs (load balance, . . .)

Outline

FIAT

Finite Element Integrator And Tabulator by Rob Kirby

http://fenicsproject.org/

FIAT understands

- Reference element shapes (line, triangle, tetrahedron)
- Quadrature rules
- Polynomial spaces
- Functionals over polynomials (dual spaces)
- Derivatives

Can build arbitrary elements by specifying the Ciarlet triple (K, P, P')

FIAT is part of the FEniCS project

FIAT

Finite Element Integrator And Tabulator by Rob Kirby

http://fenicsproject.org/

FIAT understands

- Reference element shapes (line, triangle, tetrahedron)
- Quadrature rules
- Polynomial spaces
- Functionals over polynomials (dual spaces)
- Derivatives

Can build arbitrary elements by specifying the Ciarlet triple (K, P, P')

FIAT is part of the FEniCS project

FIAT Integration

The `quadrature.fiat` file contains:

- An element (usually a family and degree) defined by FIAT
- A quadrature rule

It is run

- automatically by `make`, or
- independently by the user

It can take arguments

- `-element_family` and `-element_order`, or
- `make` takes variables `ELEMENT` and `ORDER`

Then `make` produces `quadrature.h` with:

- Quadrature points and weights
- Basis function and derivative evaluations at the quadrature points
- Integration against dual basis functions over the cell
- Local dofs for Section allocation

Outline

Outline

Outline

FErari

Finite Element rearragement to automaically reduce instructions

- Open source implementation http://www.fenics.org/wiki/FErari
- Build tensor blocks $K^{ij}_{m,m'}$ as vectors using FIAT
- Discover dependencies
  - Represented as a DAG
  - Can also use hypergraph model
- Use minimal spanning tree to construct computation

Outline

Form Decomposition

Element integrals are decomposed into analytic and geometric parts:

$$\int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) d\mathbf{x} \tag{20}$$

$$= \int_{\mathcal{T}} \frac{\partial\phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial\phi_j(\mathbf{x})}{\partial x_\alpha} d\mathbf{x} \tag{21}$$

$$= \int_{\mathcal{T}_{\text{ref}}} \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\xi_\gamma}{\partial x_\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} |J| d\mathbf{x} \tag{22}$$

$$= \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} d\mathbf{x} \tag{23}$$

$$= G^{\beta\gamma}(\mathcal{T}) K^{ij}_{\beta\gamma} \tag{24}$$

Coefficients are also put into the geometric part.

Form Decomposition

Additional fields give rise to multilinear forms.

$$\int_{\mathcal{T}} \phi_i(\mathbf{x}) \cdot \left(\phi_k(\mathbf{x}) \nabla \phi_j(\mathbf{x})\right) dA \tag{25}$$

$$= \int_{\mathcal{T}} \phi_i^\beta(\mathbf{x}) \left(\phi_k^\alpha(\mathbf{x}) \frac{\partial \phi_j^\beta(\mathbf{x})}{\partial x_\alpha}\right) dA \tag{26}$$

$$= \int_{\mathcal{T}_{\text{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} |J| dA \tag{27}$$

$$= \frac{\partial \xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} dA \tag{28}$$

$$= G^{\alpha\gamma}(\mathcal{T}) K_{\alpha\gamma}^{ijk} \tag{29}$$

The index calculus is fully developed by Kirby and Logg in A Compiler for Variational Forms.

Form Decomposition

Isoparametric Jacobians also give rise to multilinear forms

$$\int_{\mathcal{T}} \nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) dA \tag{30}$$

$$= \int_{\mathcal{T}} \frac{\partial\phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial\phi_j(\mathbf{x})}{\partial x_\alpha} dA \tag{31}$$

$$= \int_{\mathcal{T}_{\text{ref}}} \frac{\partial\xi_\beta}{\partial x_\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \frac{\partial\xi_\gamma}{\partial x_\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} |J| dA \tag{32}$$

$$= |J| \int_{\mathcal{T}_{\text{ref}}} \phi_k J_k^{\beta\alpha} \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \phi_l J_l^{\gamma\alpha} \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} dA \tag{33}$$

$$= J_k^{\beta\alpha} J_l^{\gamma\alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \phi_k \frac{\partial\phi_i(\xi)}{\partial\xi_\beta} \phi_l \frac{\partial\phi_j(\xi)}{\partial\xi_\gamma} dA \tag{34}$$

$$= G_{kl}^{\beta\gamma}(\mathcal{T}) K_{\beta\gamma}^{ijkl} \tag{35}$$

A different space could also be used for Jacobians

Element Matrix Formation

- Element matrix K is now made up of small tensors
- Contract all tensor elements with each the geometry tensor $G(\mathcal{T})$

3 0 0 0	0 -1 0 0	1 1 0 0	-4 -4 0 0	0 4 0 0	0 0 0 0
0 0 -1 0	0 0 0 3	0 0 1 1	0 0 0 0	0 0 4 0	0 0 -4 -4
1 0 1 0	0 1 0 1	3 3 3 3	-4 0 -4 0	0 0 0 0	0 -4 0 -4
-4 0 -4 0	0 0 0 0	-4 -4 0 0	8 4 4 8	0 -4 -4 -8	0 4 4 0
0 0 4 0	0 4 0 0	0 0 0 0	0 -4 -4 -8	8 4 4 8	-8 -4 -4 0
0 0 0 0	0 -4 0 -4	0 0 -4 -4	0 4 4 0	-8 -4 -4 0	8 4 4 8

Element Matrix Computation

- Element matrix K can be precomputed
  - FFC
  - SyFi
- Can be extended to nonlinearities and curved geometry
- Many redundancies among tensor elements of K
  - Could try to optimize the tensor contraction...

Abstract Problem

Given vectors $v_i \in \mathbb{R}^m$, minimize $flops(v^T g)$ for arbitrary $g \in \mathbb{R}^m$

- The set v_i is not at all random
- Not a traditional compiler optimization
- How to formulate as an optimization problem?

Outline

Complexity Reducing Relations

If $v_i^T g$ is known, is $flops(v_j^T g) < 2m - 1$?

We can use binary relations among the vectors:

- Equality
  - If $v_j = v_i$, then $flops(v_j^T g) = 0$
- Colinearity
  - If $v_j = \alpha v_i$, then $flops(v_j^T g) = 1$
- Hamming distance
  - If $dist_H(v_j, v_i) = k$, then $flops(v_j^T g) = 2k$

Algorithm for Binary Relations

- Construct a weighted graph on v_i
  - The weight $w(i,j)$ is $flops(v_j^T g)$ given $v_i^T g$
  - With the above relations, the graph is symmetric
- Find a minimum spanning tree
  - Use Prim or Kruskal for worst case $O(n^2 \log n)$
- Traverse the MST, using the appropriate calculation for each edge
  - Roots require a full dot product

Coplanarity

- Ternary relation
  - If $v_k = \alpha v_i + \beta v_j$, then $flops(v_k^T g) = 3$
  - Does not fit our undirected graph paradigm
- SVD for vector triples is expensive
  - Use a randomized projection into a few $\mathbb{R}^3$s
- Use a hypergraph?
  - MST algorithm available
- Appeal to geometry?
  - Incidence structures

Outline

Preliminary Results

Order	Entries	Base MAPs	FErari MAPs
1	6	24	7
2	21	84	15
3	55	220	45
4	120	480	176
5	231	924	443
6	406	1624	867

Outline

Modeling the Problem

- Objective is cost of dot products (tensor contractions in FEM)
  - Set of vectors V with a given arbitrary vector g
- The original MINLP has a nonconvex, nonlinear objective
- Reformulate to obtain a MILP using auxiliary binary variables

Modeling the Problem

Variables

α_{ij} = Basis expansion coefficients

y_i = Binary variable indicating membership in the basis

s_{ij} = Binary variable indicating nonzero coefficient α_{ij}

z_{ij} = Binary variable linearizes the objective function (equivalent to $y_i y_j$)

U = Upper bound on coefficients

Constraints

Eq. (36*b*) : Basis expansion

Eq. (36*c*) : Exclude nonbasis vector from the expansion

Eq. (36*d*) : Remove offdiagonal coefficients for basis vectors

Eq. (7*c*) : Exclude vanishing coefficients from cost

Original Formulation

MINLP Model

$$\text{minimize} \quad \sum_{i=1}^{n} \left\{ y_i(2m-1) + (1-y_i)\left(2\sum_{j=1, j\neq i}^{n} y_j - 1\right)\right\} \tag{36a}$$

$$\text{subject to} \quad v_i = \sum_{j=1}^{n} \alpha_{ij} v_j \qquad i = 1, \ldots, n \tag{36b}$$

$$-Uy_j \leq \alpha_{ij} \leq Uy_j \qquad i, j = 1, \ldots, n \tag{36c}$$

$$-U(1-y_i) \leq \alpha_{ij} \leq U(1-y_i) \qquad i, j = 1, \ldots, n, \tag{36d}$$

$$y_i \in \{0, 1\} \qquad i = 1, \ldots, n. \tag{36e}$$

Original Formulation

Equivalent MILP Model: $z_{ij} = y_i \cdot y_j$

$$\text{minimize} \quad 2m\sum_{i=1}^{n} y_i + 2\sum_{i=1}^{n}\sum_{j=1, j\neq i}^{n} (y_j - z_{ij}) - n \tag{36a}$$

$$\text{subject to} \quad v_i = \sum_{j=1}^{n} \alpha_{ij} v_j \qquad i = 1, \ldots, n \tag{36b}$$

$$-Uy_j \leq \alpha_{ij} \leq Uy_j \qquad i, j = 1, \ldots, n \tag{36c}$$

$$-U(1 - y_i) \leq \alpha_{ij} \leq U(1 - y_i) \qquad i, j = 1, \ldots, n,\ i \neq j \tag{36d}$$

$$z_{ij} \leq y_i, \quad z_{ij} \leq y_j, \quad z_{ij} \geq y_i + y_j - 1, \qquad i, j = 1, \ldots, n \tag{36e}$$

$$y_i \in \{0, 1\}, \quad z_{ij} \in \{0, 1\} \qquad i, j = 1, \ldots, n. \tag{36f}$$

Sparse Coefficient Formulation

- Take advantage of sparsity of α_{ij} coefficient
- Introduce binary variables s_{ij} to model existence of α_{ij}
- Add constraints $-Us_{ij} \leq \alpha_{ij} \leq Us_{ij}$

Sparse Coefficient Formulation

MINLP Model

$$\text{minimize} \quad \sum_{i=1}^{n} \left\{ y_i(2m-1) + (1-y_i)\left(2\sum_{j=1, j\neq i}^{n} s_{ij} - 1\right)\right\} \tag{37a}$$

$$\text{subject to} \quad v_i = \sum_{j=1}^{n} \alpha_{ij} v_j \qquad i = 1, \ldots, n \tag{37b}$$

$$-Us_{ij} \leq \alpha_{ij} \leq Us_{ij} \qquad i, j = 1, \ldots, n \tag{37c}$$

$$-U(1-y_i) \leq \alpha_{ij} \leq U(1-y_i) \qquad i, j = 1, \ldots, n \tag{37d}$$

$$s_{ij} \leq y_j \qquad i, j = 1, \ldots, n \tag{37e}$$

$$y_i \in \{0,1\}, \quad s_{ij} \in \{0,1\} \qquad i, j = 1, \ldots, n$$

Sparse Coefficient Formulation

Equivalent MILP Model

$$\text{minimize} \quad 2m\sum_{i=1}^{n} y_i + 2\sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n}(s_{ij} - z_{ij}) - n \tag{37a}$$

$$\text{subject to} \quad v_i = \sum_{j=1}^{n} \alpha_{ij} v_j \qquad i = 1, \ldots, n \tag{37b}$$

$$-Us_{ij} \leq \alpha_{ij} \leq Us_{ij} \qquad i, j = 1, \ldots, n \tag{37c}$$

$$-U(1 - y_i) \leq \alpha_{ij} \leq U(1 - y_i) \qquad i, j = 1, \ldots, n,\ i \neq \tag{37d}$$

$$z_{ij} \leq y_i, \quad z_{ij} \leq s_{ij}, \quad z_{ij} \geq y_i + s_{ij} - 1, \qquad i, j = 1, \ldots, n \tag{37e}$$

$$y_i \in \{0, 1\}, \quad z_{ij} \in \{0, 1\}, \quad s_{ij} \in \{0, 1\} \qquad i, j = 1, \ldots, n. \tag{37f}$$

Results

Initial Formulation

- Initial formulation only had sparsity in the α_{ij}
- MINTO was not able to produce some optimal solutions
  - Report results after 36000 seconds

Element	Default Flops	MILP Flops	MILP LPs	MILP CPU	Sparse Coef. MILP Flops	Sparse Coef. MILP LPs	Sparse Coef. MILP CPU
P_1 2D	42	42	33	0.10	34	187	0.43
P_2 2D	147	147	2577	37.12	67	6030501	36000
P_1 3D	170	166	79	0.49	146	727	3.97
P_2 3D	935	935	25283	36000	829	33200	36000

Results

Formulation with Sparse Basis

- We can also take account of the sparsity in the basis vectors
- Count only the flops for nonzero entries
  - Significantly decreases the flop count

Elements	Sparse Coefficient Flops	Sparse Basis Flops
P_1 2D	34	12
P_1 3D	146	26

Outline

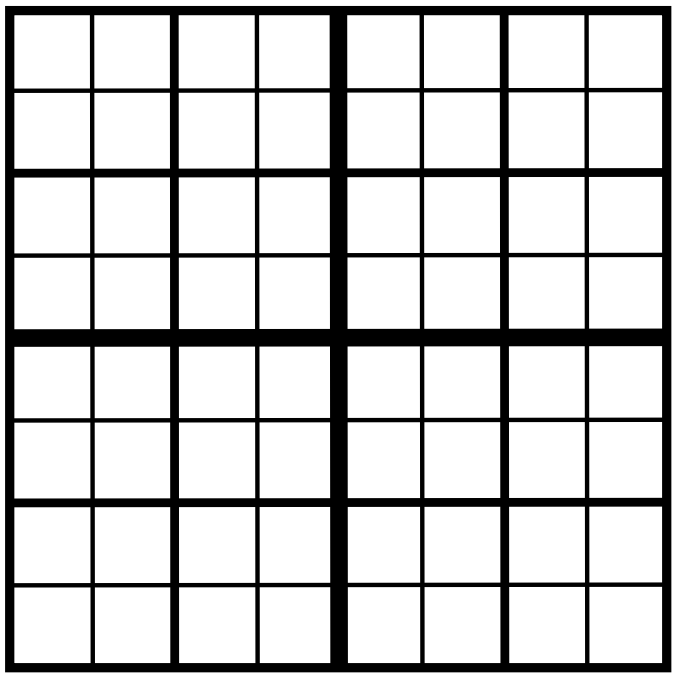
Part VI

Fast Methods

Outline

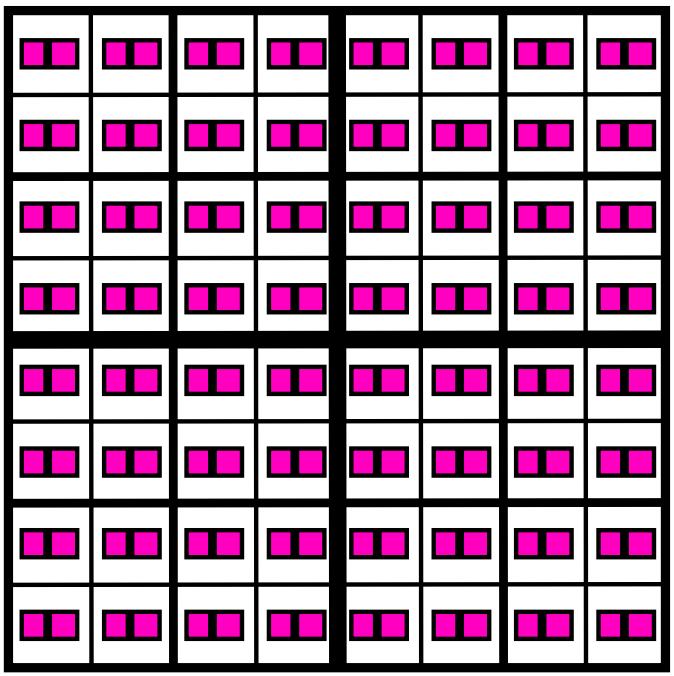
Outline

FMM in Sieve



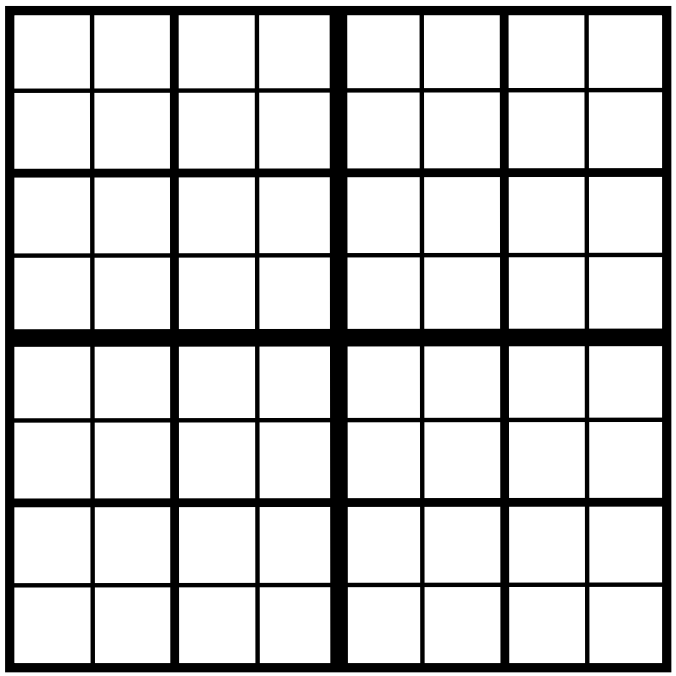
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



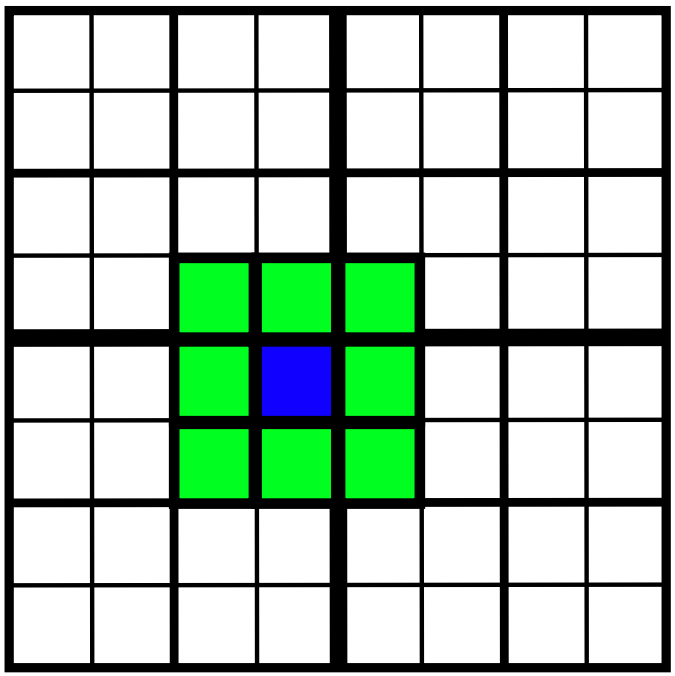
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



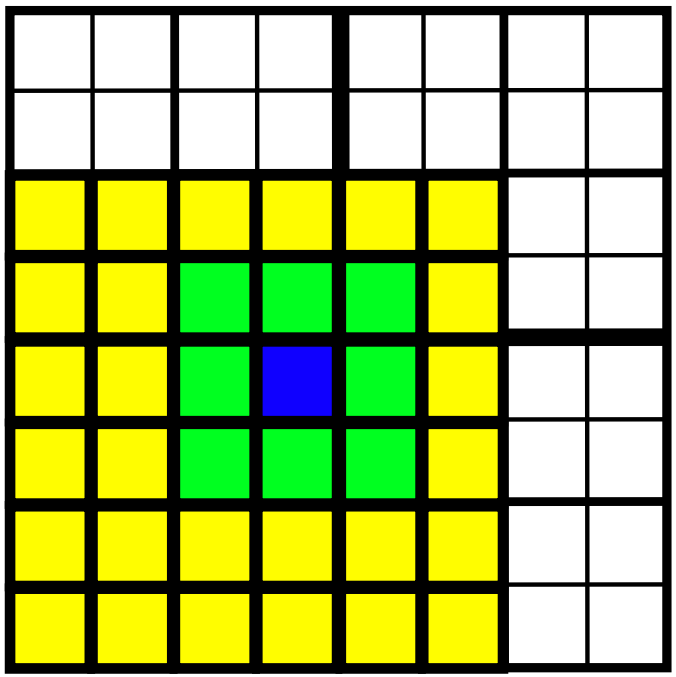
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



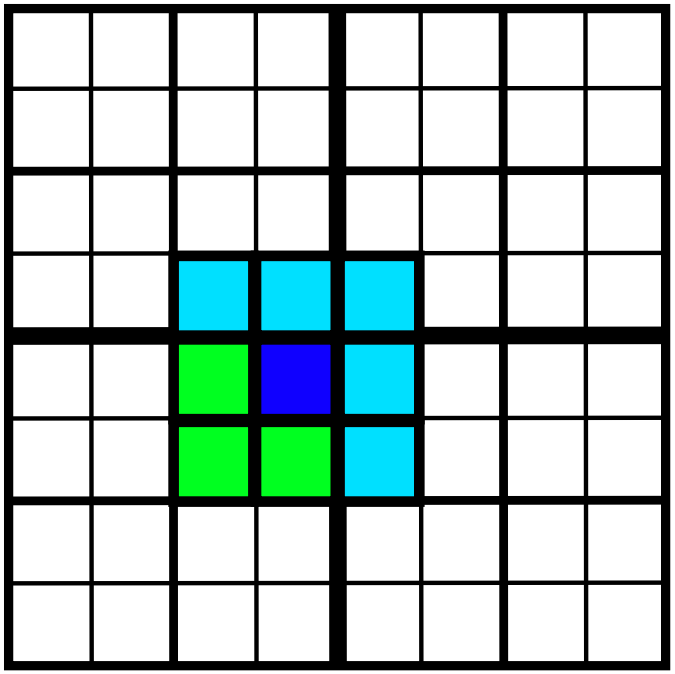
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



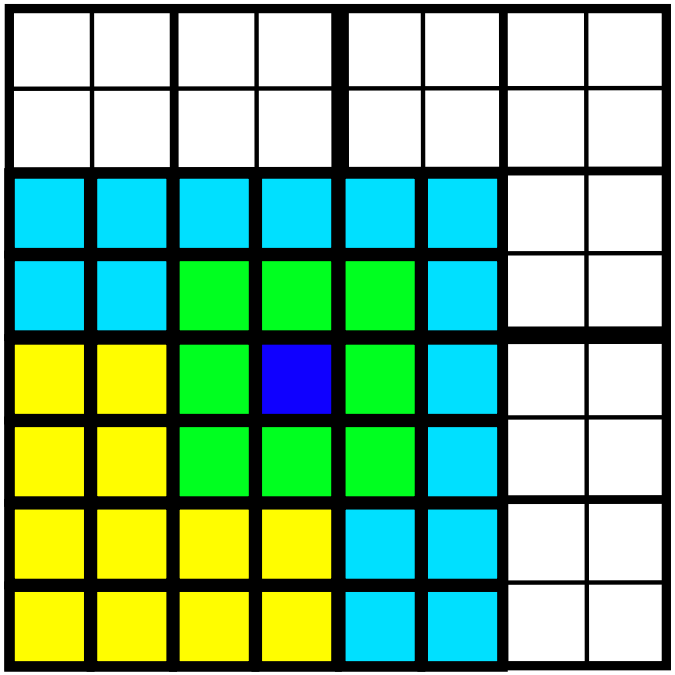
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

Quadtree Implementation

- We use binary scheme to label cells (or vertices)
- Relevant relations can be determined implicitly
  - `cone()`
  - neighbors
  - parent
  - interaction list
- When vertices are not used, we can directly connect cells
  - `cone()` becomes neighbor method

Tree Interface

- `locateBlob(blob)`
  - Locate point in the tree
- `fillNeighbors()`
  - Compute the neighbor section
- `findInteractionList()`
  - Compute the interaction list cell section, allocate value section
- `fillInteractionList(level)`
  - Compute the interaction list value section
- `fill(blobs)`
  - Compute the blob section
- `dump()`
  - Produces a verifiable repesentation of the tree

Outline

FMM Sections

FMM requires data over the Quadtree distributed by:

- box
  - Box centers, Neighbors
- box + neighbors
  - Blobs
- box + interaction list
  - Interaction list cells and values
  - Multipole and local coefficients

FMM Sections

FMM requires data over the Quadtree distributed by:

- box
  - Box centers, Neighbors
- box + neighbors
  - Blobs
- box + interaction list
  - Interaction list cells and values
  - Multipole and local coefficients

FMM Sections

FMM requires data over the Quadtree distributed by:

- box
  - Box centers, Neighbors
- box + neighbors
  - Blobs
- box + interaction list
  - Interaction list cells and values
  - Multipole and local coefficients

FMM Sections

FMM requires data over the Quadtree distributed by:

- box
  - Box centers, Neighbors
- box + neighbors
  - Blobs
- box + interaction list
  - Interaction list cells and values
  - Multipole and local coefficients

Notice this is multiscale since data is divided at each level

Outline

Evaluator Interface

- `initializeExpansions(tree, blobInfo)`
  - Generate multipole expansions on the lowest level
  - Requires loop over cells
  - $O(p)$
- `upwardSweep(tree)`
  - Translate multipole expansions to intermediate levels
  - Requires loop over cells and children (support)
  - $O(p^2)$
- `downwardSweep(tree)`
  - Convert multipole to local expansions and translate local expansions on intermediate levels
  - Requires loop over cells and parent (cone)
  - $O(p^2)$

Evaluator Interface

- `evaluateBlobs(tree, blobInfo)`
  - Evaluate direct and local field interactions on lowest level
  - Requires loop over cells and neighbors (in section)
  - $O(p^2)$
- `evaluate(tree, blobs, blobInfo)`
  - Calculate the complete interaction (multipole + direct)

Kernel Interface

Method	Description
`P2M(t)`	Multipole expansion coefficients
`L2P(t)`	Local expansion coefficients
`M2M(t)`	Multipole-to-multipole translation
`M2L(t)`	Multipole-to-local translation
`L2L(t)`	Local-to-local translation
`evaluate(blobs)`	Direct interaction

- `Evaluator` is templated over `Kernel`
- There are alternative kernel-independent methods
  - kifmm3d

Outline

Parallel Tree Implementation

- Divide tree into a root and local trees
- Distribute local trees among processes
- Provide communication pattern for local sections (overlap)
  - Both neighbor and interaction list overlaps
  - Sieve generates MPI from high level description

Parallel Tree Implementation

How should we distribute trees?

- Multiple local trees per process allows good load balance
- Partition weighted graph
  - Minimize load imbalance and communication
  - Computation estimate:

    Leaf $N_i p$ (P2M) + $n_l p^2$ (M2L) + $N_i p$ (L2P) + $3^d N_i^2$ (P2P)

    Interior $n_c p^2$ (M2M) + $n_l p^2$ (M2L) + $n_c p^2$ (L2L)

  - Communication estimate:

    Diagonal $n_c(L - k - 1)$

    Lateral $2^d \frac{2^{m(L-k-1)}-1}{2^m-1}$ for incidence dimesion m
- Leverage existing work on graph partitioning
  - ParMetis

Parallel Tree Implementation

Why should a good partition exist?

Shang-hua Teng, Provably good partitioning and load balancing algorithms for parallel adaptive N-body simulation, SIAM J. Sci. Comput., **19**(2), 1998.

- Good partitions exist for non-uniform distributions
  - 2D $\mathcal{O}\left(\sqrt{n}(\log n)^{3/2}\right)$ edgecut
  - 3D $\mathcal{O}\left(n^{2/3}(\log n)^{4/3}\right)$ edgecut
- As scalable as regular grids
- As efficient as uniform distributions
- ParMetis will find a nearly optimal partition

Parallel Tree Implementation

Will ParMetis find it?

George Karypis and Vipin Kumar, Analysis of Multilevel Graph Partitioning, Supercomputing, 1995.

- Good partitions exist for non-uniform distributions
  - 2D $C_i = 1.24^i C_0$ for random matching
  - 3D $C_i = 1.21^i C_0$?? for random matching
- 3D proof needs assurance that averge degree does not increase
- Efficient in practice

Parallel Tree Implementation
Advantages

- Simplicity
- Complete serial code reuse
- Provably good performance and scalability

Parallel Tree Implementation
Advantages

- Simplicity
- Complete serial code reuse
- Provably good performance and scalability

Parallel Tree Implementation
Advantages

- Simplicity
- Complete serial code reuse
- Provably good performance and scalability

Parallel Tree Interface

- `fillNeighbors()`
  - Compute neighbor overlap, and send neighbors
- `findInteractionList()`
  - Compute the interaction list overlap
- `fillInteractionList(level)`
  - Complete and copy into local interaction sections
- `fill(blobs)`
  - Now must scatter blobs to local trees
  - Uses `scatterBlobs()` and `gatherBlobs()`

Parallel Data Movement

1. Complete neighbor section
2. Upward sweep
   1. Upward sweep on local trees
   2. Gather to root tree
   3. Upward sweep on root tree
3. Complete interaction list section
4. Downward sweep
   1. Downward sweep on root tree
   2. Scatter to local trees
   3. Downward sweep on local trees

Parallel Evaluator Interface

- `initializeExpansions(local trees, blobInfo)`
  - Evaluate each local tree
- `upwardSweep(local trees, partition, root tree)`
  - Evaluate each local tree and then gather to root tree
- `downwardSweep(local trees, partition, root tree)`
  - Scatter from root tree and then evaluate each local tree
- `evaluateBlobs(local trees, blobInfo)`
  - Evaluate on all local trees
- `evaluate(tree, blobs, blobInfo)`
  - Identical

Outline

Recursive Parallel

- For large problems, a single root can be a bottleneck
- We can recursively assign roots to subtrees
  - Bandwidth to root is controlled
  - What about utilization?
- Root computation is similar to MG coarse solve

Outline

Outline

A DMDA is more than a Mesh

A DMDA contains topology, geometry, and (sometimes) an implicit Q1 discretization.

It is used as a template to create

- Vectors (functions)
- Matrices (linear operators)

DMDA Global vs. Local Numbering

- **Global**: Each vertex has a unique id belongs on a unique process
- **Local**: Numbering includes vertices from neighboring processes
  - These are called ghost vertices

Proc 2			Proc 3	
X	X	X	X	X
X	X	X	X	X
12	13	14	15	X
8	9	10	11	X
4	5	6	7	X
0	1	2	3	X
Proc 0			Proc 1	

Local numbering

Proc 2			Proc 3	
21	22	23	28	29
18	19	20	26	27
15	16	17	24	25
6	7	8	13	14
3	4	5	11	12
0	1	2	9	10
Proc 0			Proc 1	

Global numbering

DMDA Local Function

User provided function calculates the nonlinear residual (in 2D)

```
(* lf )(DMDALocalInfo *info, PetscScalar**x, PetscScalar **r, void *ctx)
```

`info`: All layout and numbering information

`x`: The current solution (a multidimensional array)

`r`: The residual

`ctx`: The user context passed to `DMDASNESSetFunctionLocal()`

The local DMDA function is activated by calling

```
DMDASNESSetFunctionLocal(dm, INSERT_VALUES, lfunc, &ctx)
```

Bratu Residual Evaluation

$$\Delta u + \lambda e^u = 0$$

```
ResLocal(DMDALocalInfo *info, PetscScalar **x, PetscScalar **f, void *ctx)
for(j = info->ys; j < info->ys+info->ym; ++j) {
  for(i = info->xs; i < info->xs+info->xm; ++i) {
    u = x[j][i];
    if (i==0 || j==0 || i == M || j == N) {
      f[j][i] = 2.0*(hydhx+hxdhy)*u; continue;
    }
    u_xx     = (2.0*u - x[j][i-1] - x[j][i+1])*hydhx;
    u_yy     = (2.0*u - x[j-1][i] - x[j+1][i])*hxdhy;
    f[j][i] = u_xx + u_yy - hx*hy*lambda*exp(u);
}}}
```

$PETSC_DIR/src/snes/examples/tutorials/ex5.c

DMDA Local Jacobian

User provided function calculates the Jacobian (in 2D)

```
(* ljac )(DMDALocalInfo *info, PetscScalar**x, Mat J, void *ctx)
```

`info`: All layout and numbering information

`x`: The current solution

`J`: The Jacobian

`ctx`: The user context passed to DASetLocalJacobian()

The local DMDA function is activated by calling

```
DMDASNESSetJacobianLocal(dm, ljac, &ctx)
```

Setting Values on Regular Grids

PETSc provides

```
MatSetValuesStencil(Mat A, m, MatStencil idxm[], n, MatStencil idxn[],
                    PetscScalar values[], InsertMode mode)
```

- Each row or column is actually a **MatStencil**
  - This specifies grid coordinates and a component if necessary
  - Can imagine for unstructured grids, they are *vertices*
- The values are the same logically dense block in row/col

Updating Ghosts

Two-step process enables overlapping computation and communication

- DMGlobalToLocalBegin(da, gvec, mode, lvec)
  - `gvec` provides the data
  - `mode` is either `INSERT_VALUES` or `ADD_VALUES`
  - `lvec` holds the local and ghost values
- DMGlobalToLocalEnd(da, gvec, mode, lvec)
  - Finishes the communication

The process can be reversed with DALocalToGlobalBegin/End().

DM Integration with SNES

- DM supplies global residual and Jacobian to SNES
  - User supplies local version to DM
  - The `Rhs_*()` and `Jac_*()` functions in the example
- Allows automatic parallelism
- Allows grid hierarchy
  - Enables multigrid once interpolation/restriction is defined
- Paradigm is developed in unstructured work
  - Solve needs scatter into contiguous global vectors (initial guess)
- Handle Neumann BC using `KSPSetNullSpace()`

Multigrid with DM

Allows multigrid with some simple command line options

- `-pc_type mg`, `-pc_mg_levels`
- `-pc_mg_type`, `-pc_mg_cycle_type`, `-pc_mg_galerkin`
- `-mg_levels_1_ksp_type`, `-mg_levels_1_pc_type`
- `-mg_coarse_ksp_type`, `-mg_coarse_pc_type`
- `-da_refine`, `-ksp_view`

Interface also works with GAMG and 3rd party packages like ML

Outline

Unstructured Meshes

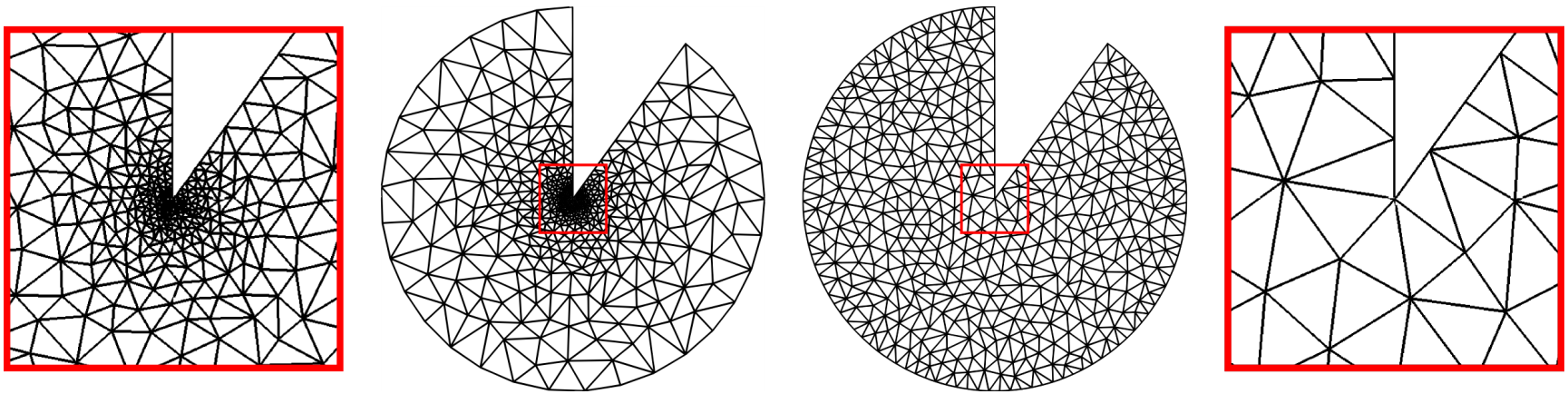
- Same `DMMG` options as the structured case
- Mesh refinement
  - Ruppert algorithm in Triangle and TetGen
- Mesh coarsening
  - Talmor-Miller algorithm in PETSc
- More advanced options
  - `-dmmg_refine`
  - `-dmmg_hierarchy`
- Current version only works for linear elements

A Priori refinement

For the Poisson problem, meshes with reentrant corners have a length-scale requirement in order to maintain accuracy:

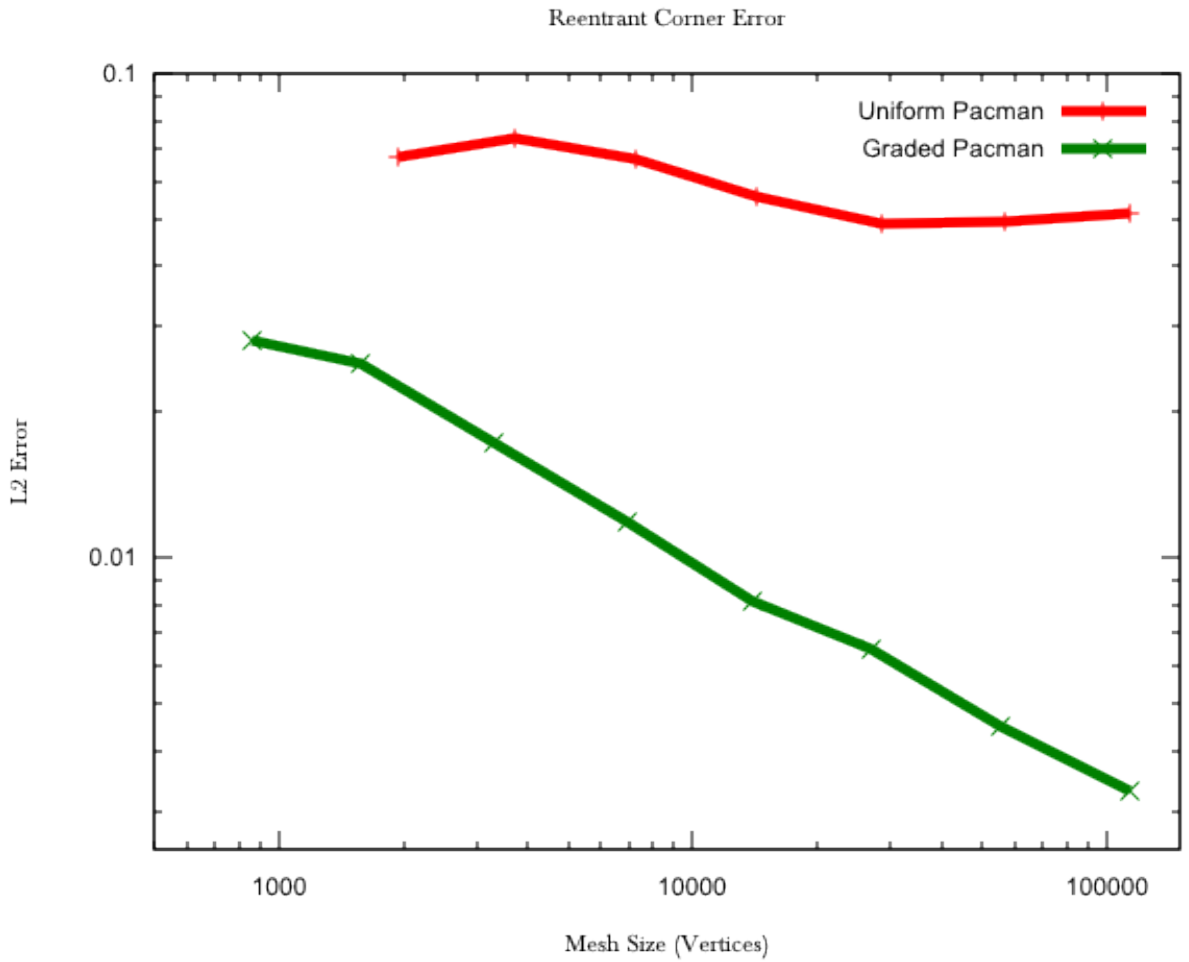
$$C_{low} r^{1-\mu} \leq h \leq C_{high} r^{1-\mu}$$

$$\mu \leq \frac{\pi}{\theta}$$



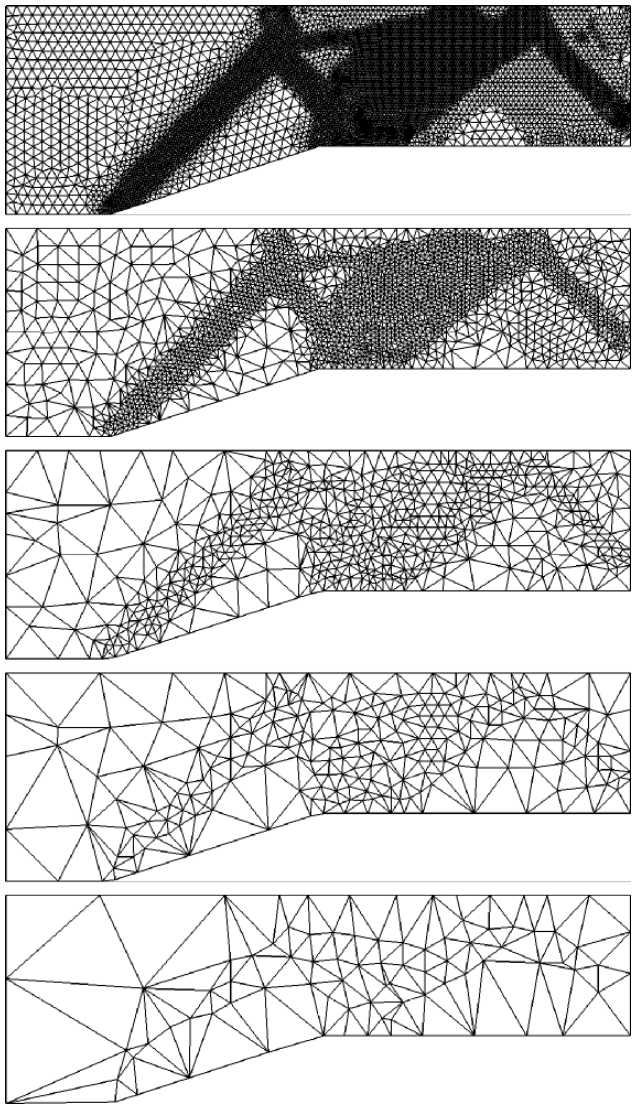
The Folly of Uniform Refinement

uniform refinement may fail to eliminate error



Geometric Multigrid

- We allow the user to refine for fidelity
- Coarse grids are created automatically
- Could make use of AMG interpolation schemes



Requirements of Geometric Multigrid

- Sufficient conditions for optimal-order convergence:
  - $|M_c| < 2|M_f|$ in terms of cells
  - any cell in M_c overlaps a bounded # of cells in M_f
  - monotonic increase in cell length-scale
- Each M_k satisfies the **quasi-uniformity** condition:

$$C_1 h_k \leq h_K \leq C_2 \rho_K$$

  - h_K is the length-scale (longest edge) of any cell K
  - h_k is the maximum length-scale in the mesh M_k
  - ρ_K is the diameter of the inscribed ball in K

Requirements of Geometric Multigrid

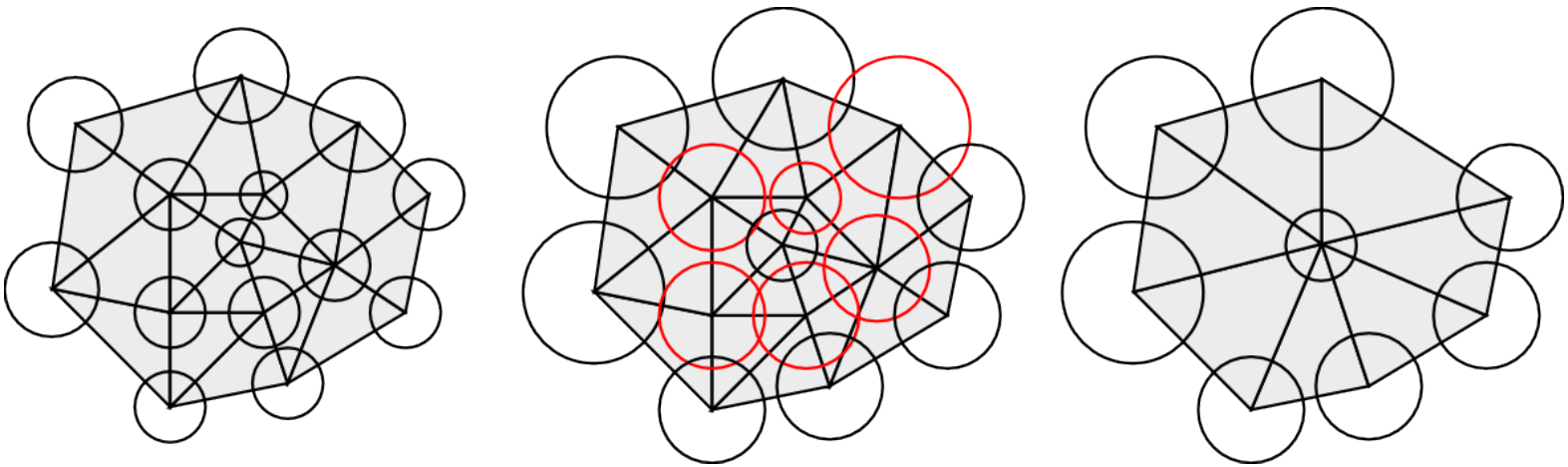
- Sufficient conditions for optimal-order convergence:
  - $|M_c| < 2|M_f|$ in terms of cells
  - any cell in M_c overlaps a bounded # of cells in M_f
  - monotonic increase in cell length-scale
- Each M_k satisfies the **quasi-uniformity** condition:

$$C_1 h_k \leq h_K \leq C_2 \rho_K$$

  - h_K is the length-scale (longest edge) of any cell K
  - h_k is the maximum length-scale in the mesh M_k
  - ρ_K is the diameter of the inscribed ball in K

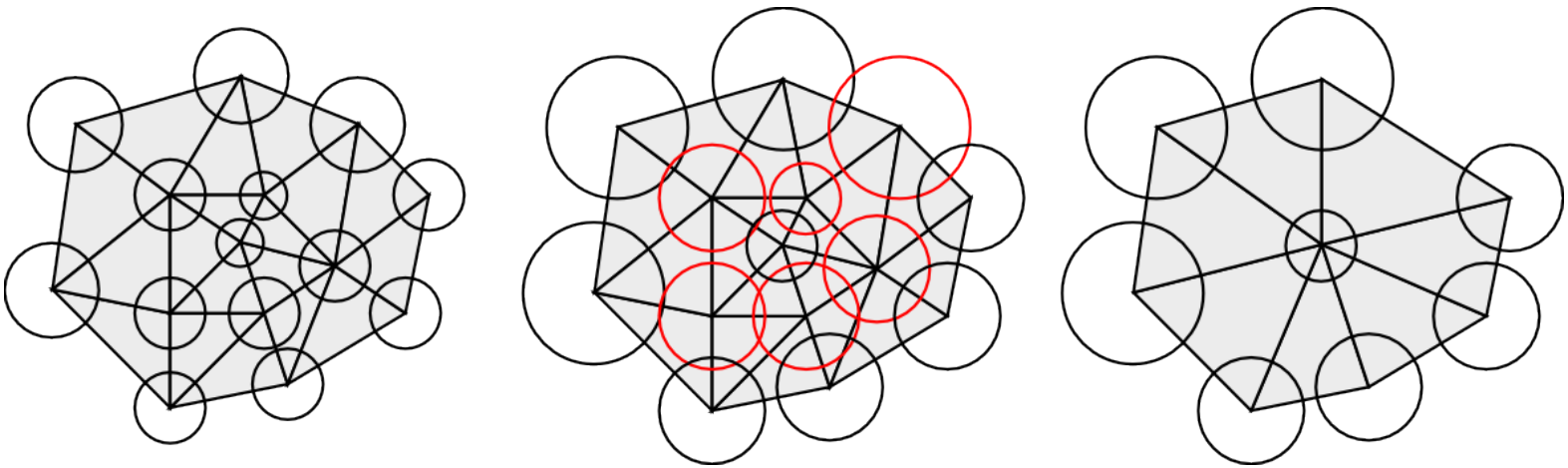
Function Based Coarsening

- (Miller, Talmor, Teng; 1997)
- triangulated planar graphs $\equiv$ disk-packings (Koebe; 1934)
- define a spacing function S() over the vertices
- obvious one: $S(v) = \frac{dist(NN(v),v)}{2}$



Function Based Coarsening

- pick a subset of the vertices such that $\beta(S(v) + S(w)) > dist(v, w)$
- for all $v, w \in M$, with $\beta > 1$
- dimension independent
- provides guarantees on the size/quality of the resulting meshes



Decimation Algorithm

- Loop over the vertices
  - include a vertex in the new mesh
  - remove colliding adjacent vertices from the mesh
  - remesh *links* of removed vertices
  - repeat until no vertices are removed.
- Eventually we have that
  - **every** vertex is either included or removed
  - bounded degree mesh $\Rightarrow O(n)$ time
- Remeshing may be performed either during or after coarsening
  - local Delaunay remeshing can be done in 2D and 3D
  - faster to connect edges and remesh later

Decimation Algorithm

- Loop over the vertices
  - include a vertex in the new mesh
  - remove colliding adjacent vertices from the mesh
  - remesh *links* of removed vertices
  - repeat until no vertices are removed.
- Eventually we have that
  - **every** vertex is either included or removed
  - bounded degree mesh $\Rightarrow O(n)$ time
- Remeshing may be performed either during or after coarsening
  - local Delaunay remeshing can be done in 2D and 3D
  - faster to connect edges and remesh later

Decimation Algorithm

- Loop over the vertices
  - include a vertex in the new mesh
  - remove colliding adjacent vertices from the mesh
  - remesh *links* of removed vertices
  - repeat until no vertices are removed.
- Eventually we have that
  - **every** vertex is either included or removed
  - bounded degree mesh $\Rightarrow O(n)$ time
- Remeshing may be performed either during or after coarsening
  - local Delaunay remeshing can be done in 2D and 3D
  - faster to connect edges and remesh later

Decimation Algorithm

- Loop over the vertices
  - include a vertex in the new mesh
  - remove colliding adjacent vertices from the mesh
  - remesh *links* of removed vertices
  - repeat until no vertices are removed.
- Eventually we have that
  - **every** vertex is either included or removed
  - bounded degree mesh $\Rightarrow O(n)$ time
- Remeshing may be performed either during or after coarsening
  - local Delaunay remeshing can be done in 2D and 3D
  - faster to connect edges and remesh later

Decimation Algorithm

- Loop over the vertices
  - include a vertex in the new mesh
  - remove colliding adjacent vertices from the mesh
  - remesh *links* of removed vertices
  - repeat until no vertices are removed.
- Eventually we have that
  - **every** vertex is either included or removed
  - bounded degree mesh $\Rightarrow O(n)$ time
- Remeshing may be performed either during or after coarsening
  - local Delaunay remeshing can be done in 2D and 3D
  - faster to connect edges and remesh later

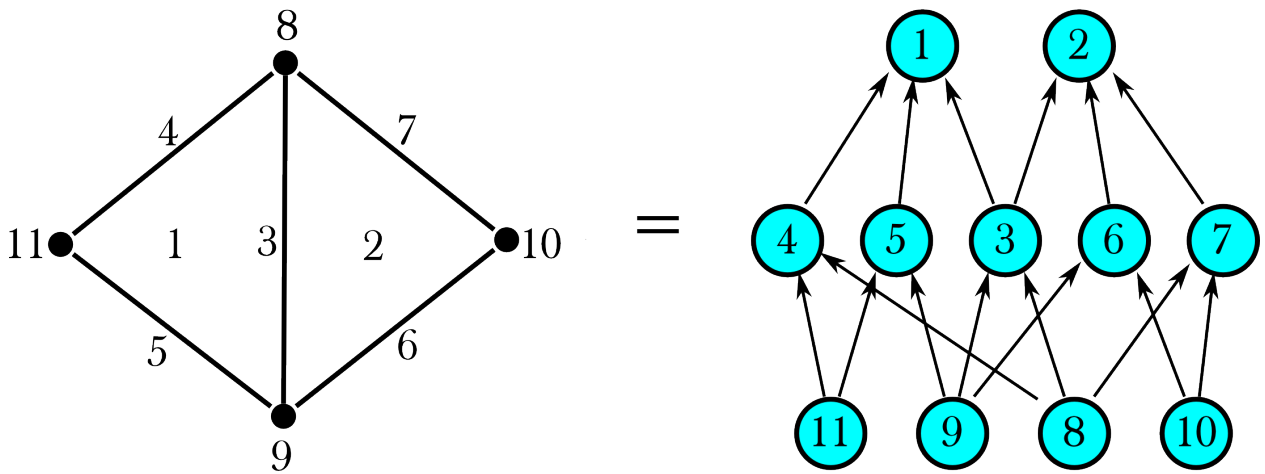
Decimation Algorithm

- Loop over the vertices
  - include a vertex in the new mesh
  - remove colliding adjacent vertices from the mesh
  - remesh *links* of removed vertices
  - repeat until no vertices are removed.
- Eventually we have that
  - **every** vertex is either included or removed
  - bounded degree mesh $\Rightarrow O(n)$ time
- Remeshing may be performed either during or after coarsening
  - local Delaunay remeshing can be done in 2D and 3D
  - faster to connect edges and remesh later

Implementation in *Sieve*

Peter Brune, 2008

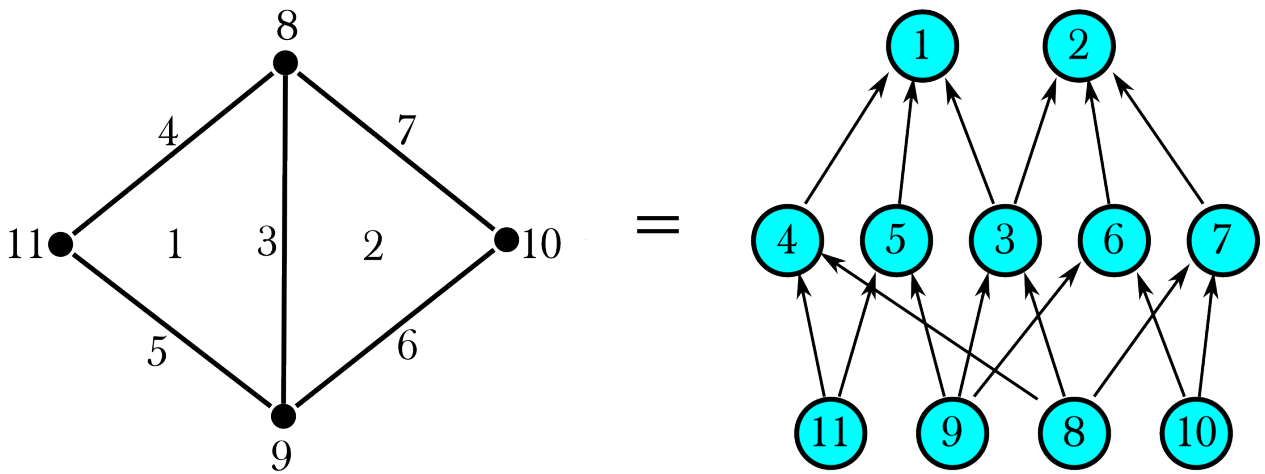
- vertex neighbors: $cone(support(v)) \setminus v$
- vertex link: $closure(star(v)) \setminus star(closure(v))$
- connectivity graph induced by limiting sieve depth
- remeshing can be handled as local modifications on the sieve
- meshing operations, such as *cone construction* easy



Implementation in *Sieve*

Peter Brune, 2008

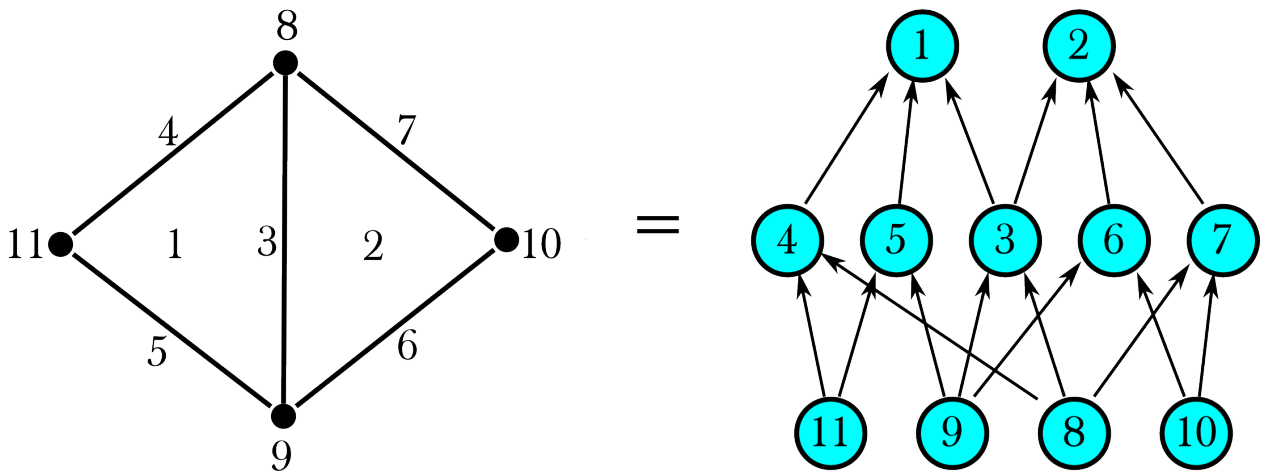
- vertex neighbors: $cone(support(v)) \setminus v$
- vertex link: $closure(star(v)) \setminus star(closure(v))$
- connectivity graph induced by limiting sieve depth
- remeshing can be handled as local modifications on the sieve
- meshing operations, such as *cone construction* easy



Implementation in *Sieve*

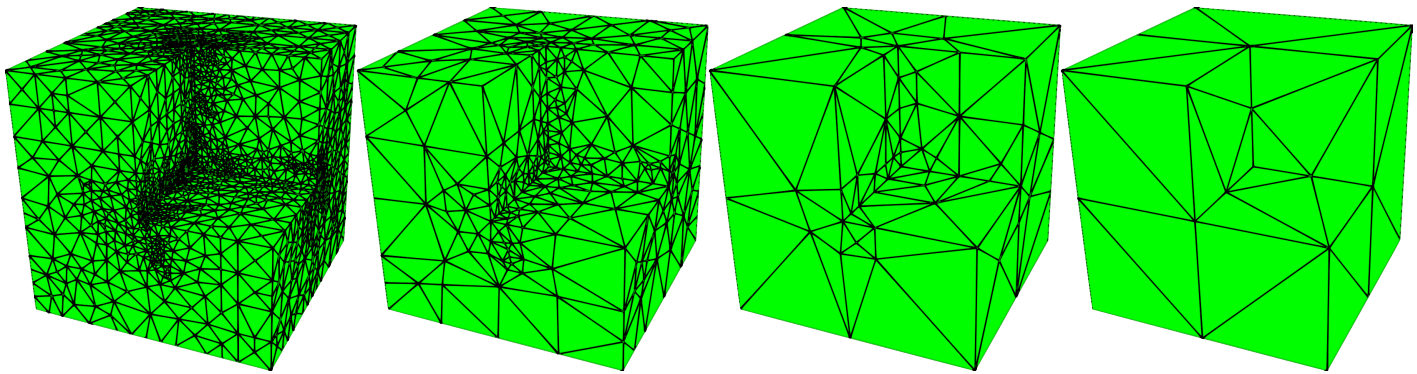
Peter Brune, 2008

- vertex neighbors: $cone(support(v)) \setminus v$
- vertex link: $closure(star(v)) \setminus star(closure(v))$
- connectivity graph induced by limiting sieve depth
- remeshing can be handled as local modifications on the sieve
- meshing operations, such as *cone construction* easy



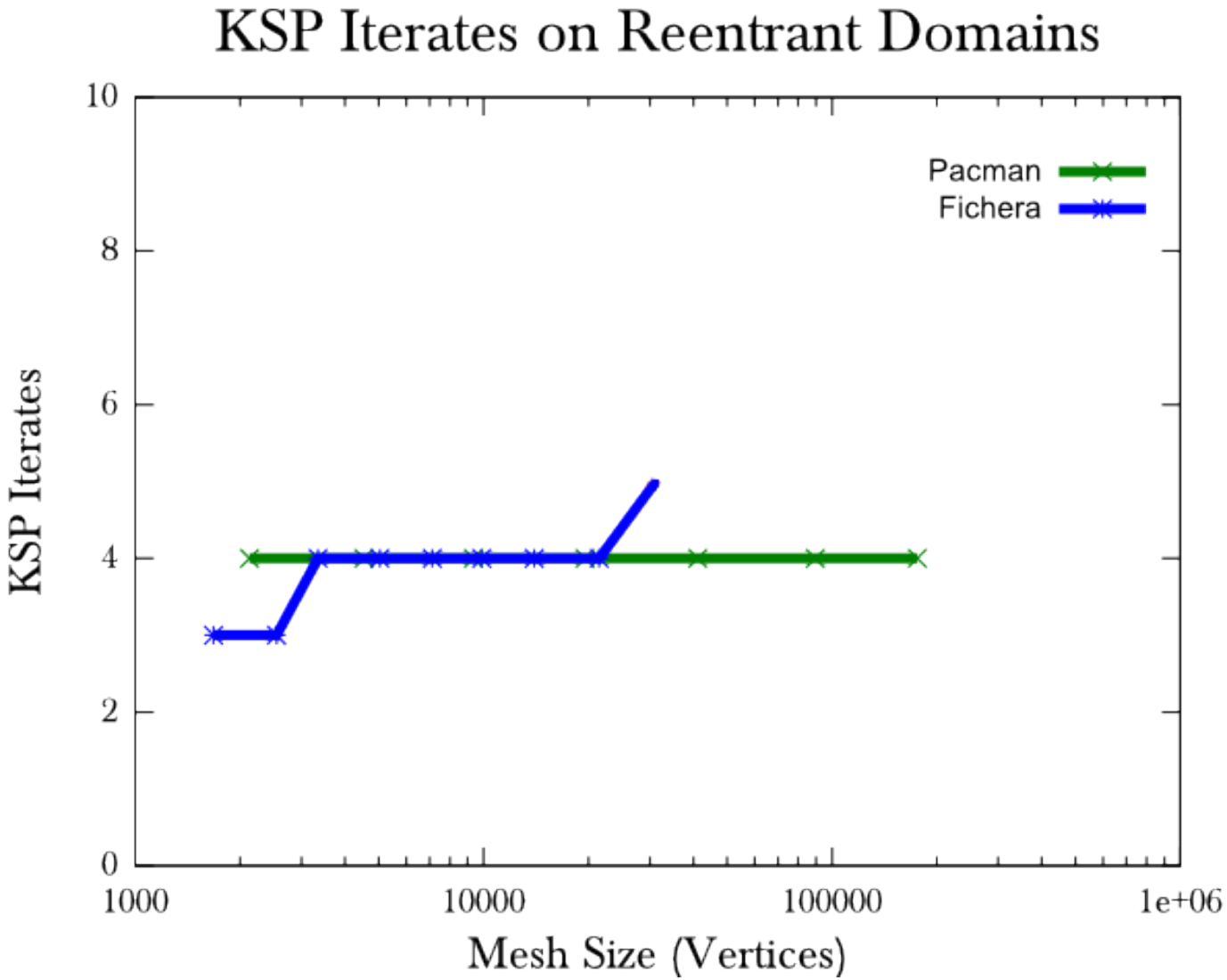
3D Test Problem

- Reentrant corner
- $-\Delta u = f$
- $f(x, y, z) = 3\sin(x + y + z)$
- Exact Solution: $u(x, y, z) = \sin(x + y + z)$



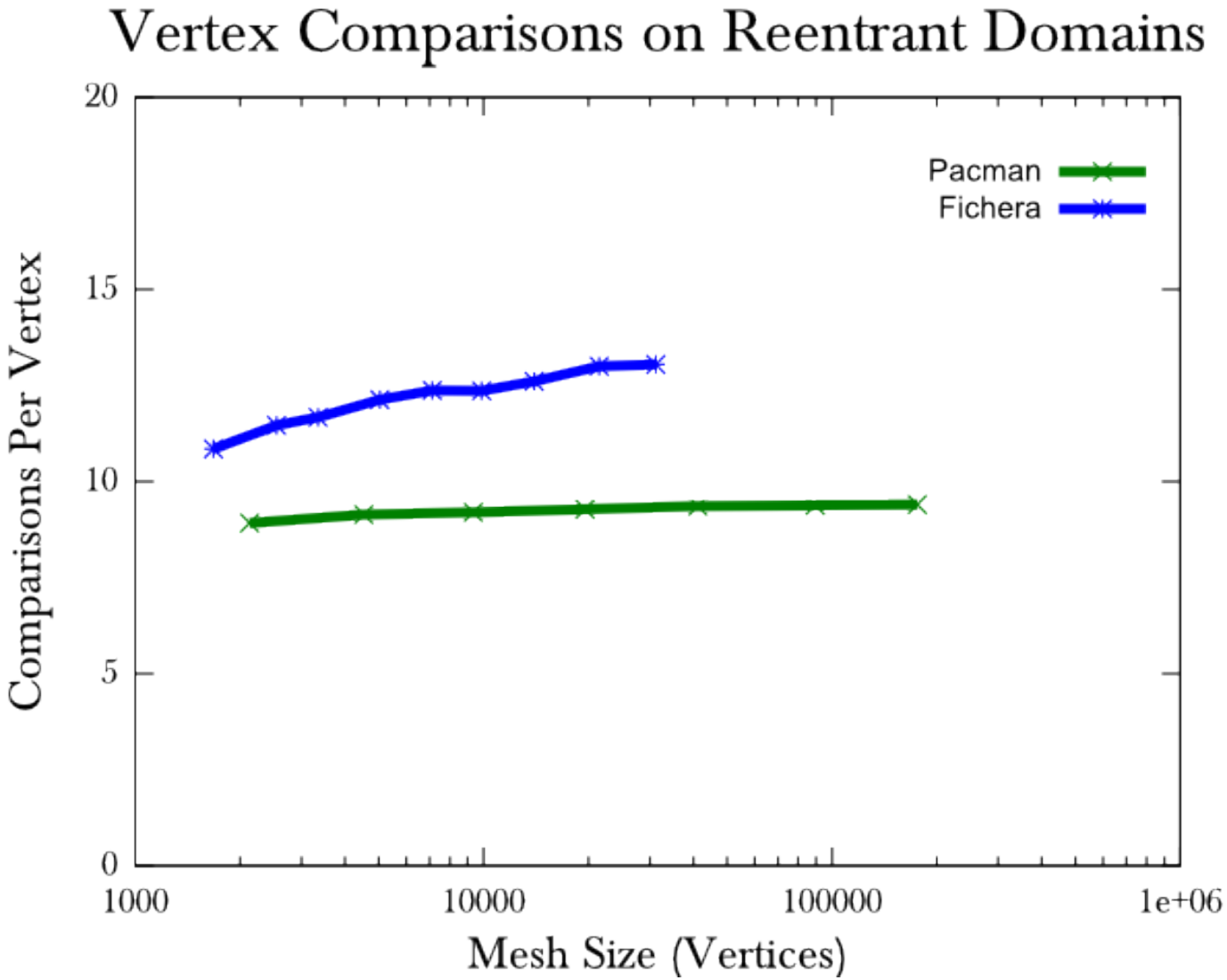
GMG Performance

Linear solver iterates are nearly as system size increases:



GMG Performance

Coarsening work is nearly constant as system size increases:



Quality Experiments

Table: Hierarchy quality metrics - 2D

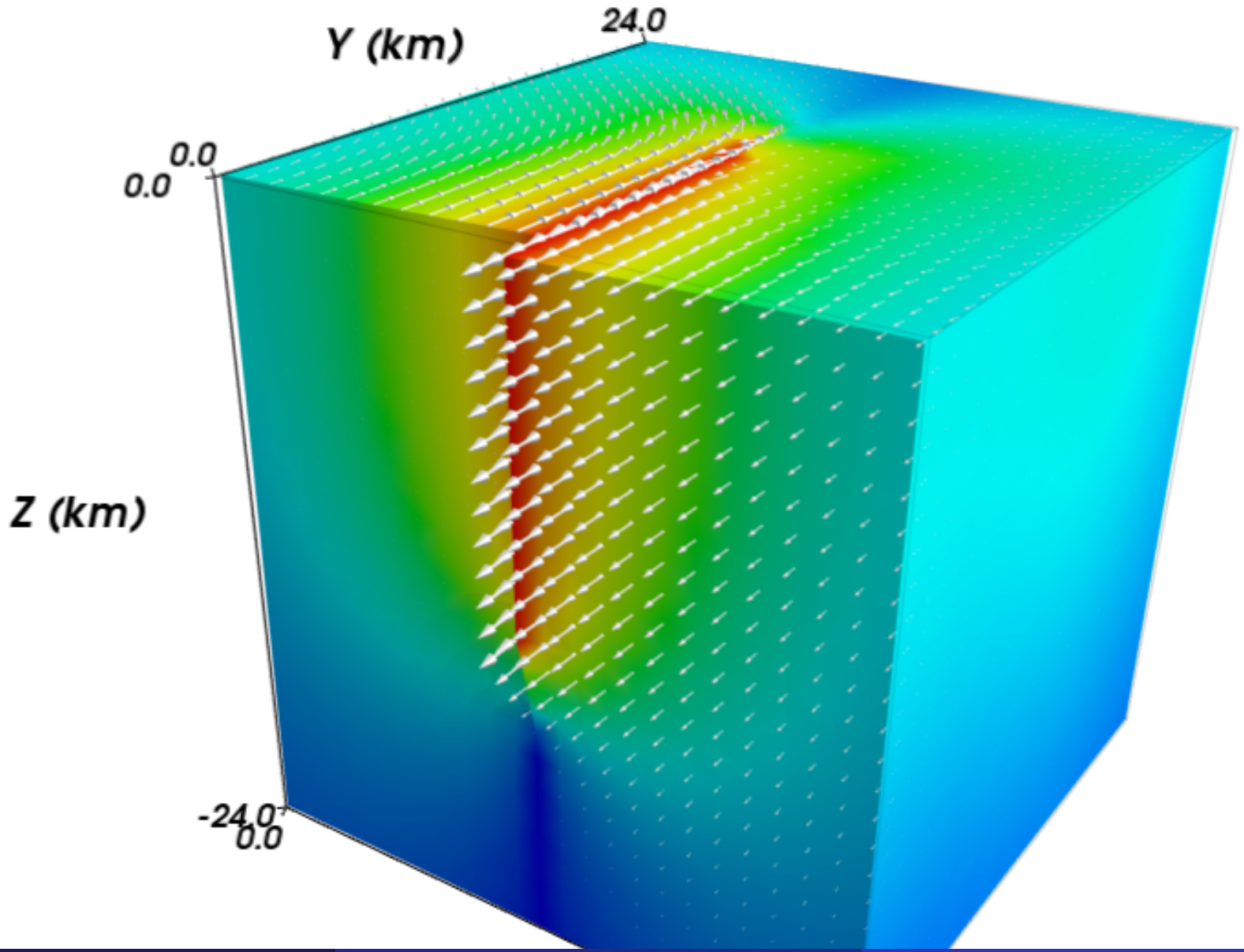
Pacman Mesh, $\beta = 1.45$						
level	cells	vertices	$\frac{min(h_K)}{h_k}$	max $\frac{h_K}{\rho_k}$	min(h_K)	max. overlap
0	19927	10149	0.020451	4.134135	0.001305	-
1	5297	2731	0.016971	4.435928	0.002094	23
2	3028	1572	0.014506	4.295703	0.002603	14
3	1628	856	0.014797	5.295322	0.003339	14
4	863	464	0.011375	6.403574	0.003339	14
5	449	250	0.022317	6.330512	0.007979	13

Part VII

Sample Application: Fault Mechanics

Outline

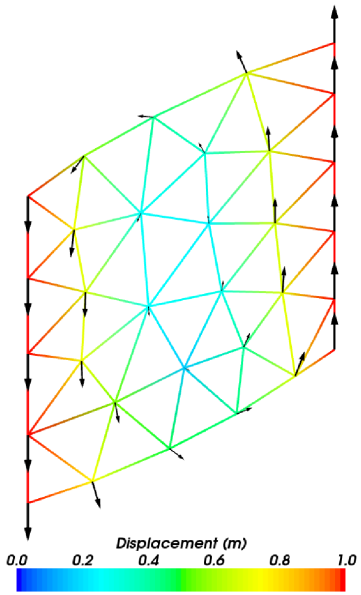
Reverse-slip Benchmark

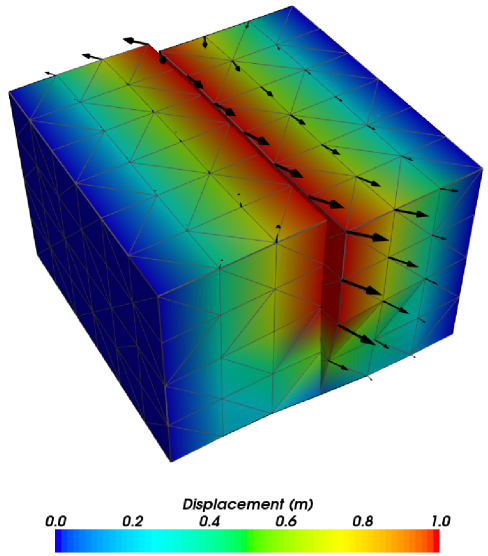


Outline

Multiple Mesh Types

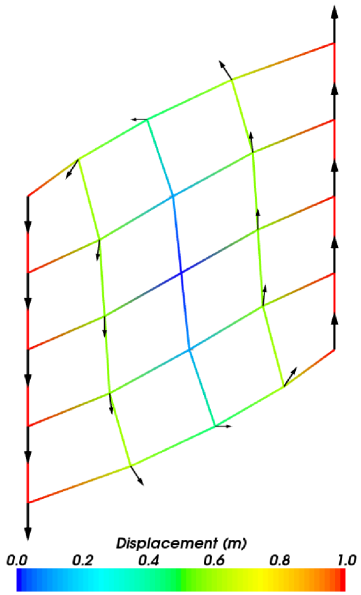
Triangular

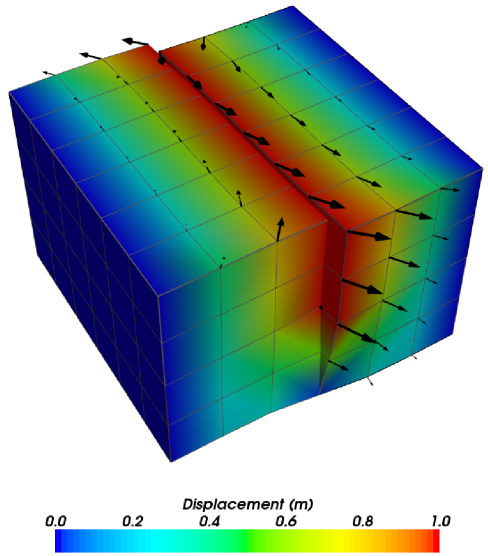




Tetrahedral

Rectangular





Hexahedral

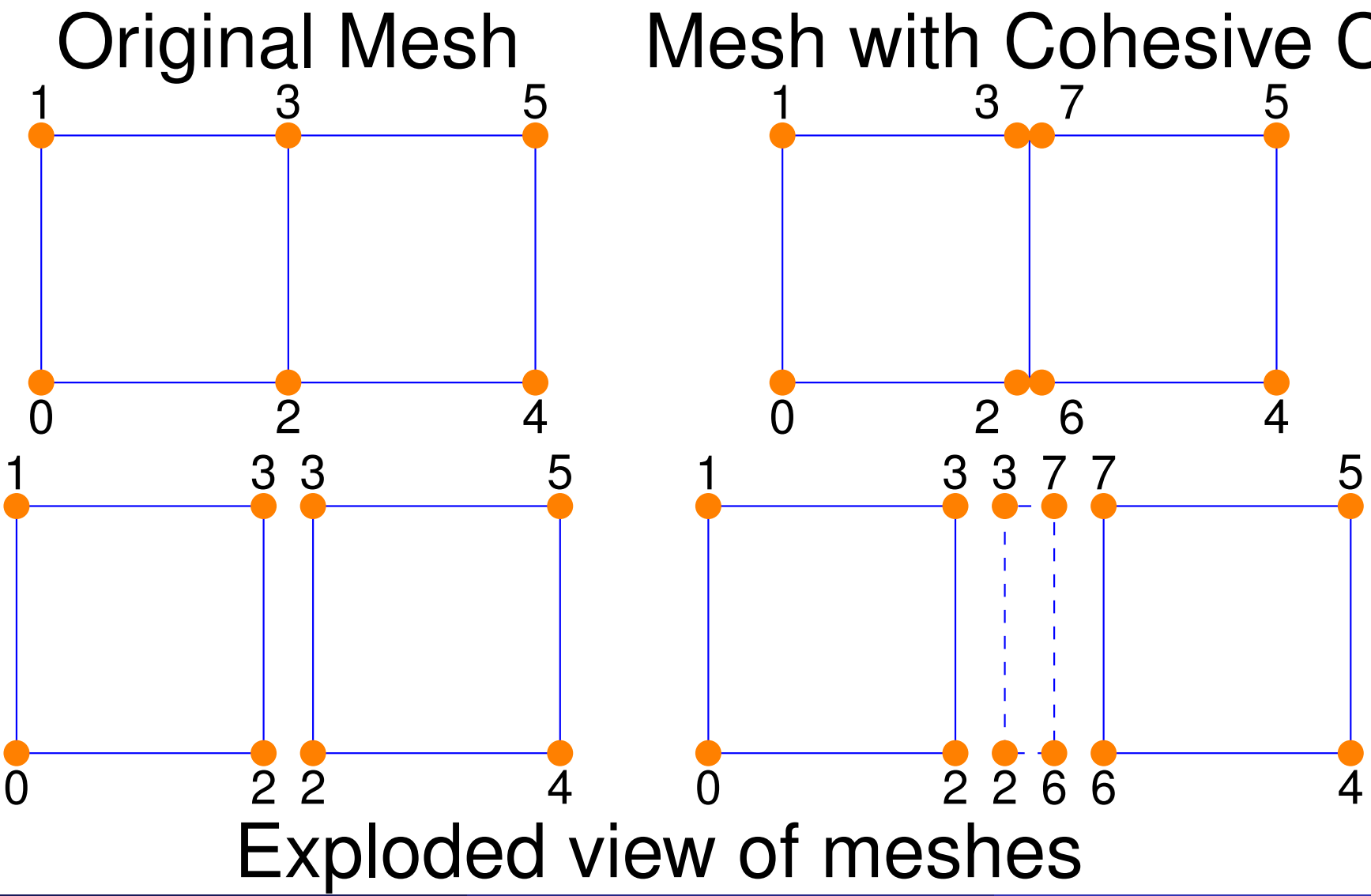
Outline

Parallelism

- Function and Operator Assembly
  - Parallel element integration over multiple materials/models
  - Assembly uses completion for functions and PETSc Mat for operators
- Algebraic sovlers
  - Use MUMPS for small problems
  - PETSc ASM/ILU for large problems
  - Hope to use unstructured MG when fault support is implemented
- Parallel data movement routines do not change for
  - Different dimension
  - Different element shapes
  - Different discretization
  - Fault inclusion

Outline

Cohesive Cells



Cohesive Cells

Cohesive cells are used to enforce slip conditions on a fault

- Demand complex mesh manipulation
  - We allow specification of only fault vertices
  - Must “sew” together on output
- Use Lagrange multipliers to enforce constraints
  - Forces illuminate physics
- Allow different fault constitutive models
  - Simplest is enforced slip
  - Now have fault constitutive models

Splitting the Mesh

- In order to create a fault, the generator provides
  - a set of fault vertices, or
  - a set of fault faces.
- Fault vertices, unlike fault faces, must be
  - combined into faces on a fault mesh, and
  - oriented
- The fault mesh is used to
  - split vertices along the fault
  - introduce prism elements between adjacent fault faces
- Sieve code works for
  - any dimension
  - any element shape

Splitting the Mesh

- In order to create a fault, the generator provides
  - a set of fault vertices, or
  - a set of fault faces.
- Fault vertices, unlike fault faces, must be
  - combined into faces on a fault mesh, and
  - oriented
- The fault mesh is used to
  - split vertices along the fault
  - introduce prism elements between adjacent fault faces
- Sieve code works for
  - any dimension
  - any element shape

Splitting the Mesh

- In order to create a fault, the generator provides
  - a set of fault vertices, or
  - a set of fault faces.
- Fault vertices, unlike fault faces, must be
  - combined into faces on a fault mesh, and
  - oriented
- The fault mesh is used to
  - split vertices along the fault
  - introduce prism elements between adjacent fault faces
- Sieve code works for
  - any dimension
  - any element shape

Splitting the Mesh

- In order to create a fault, the generator provides
  - a set of fault vertices, or
  - a set of fault faces.
- Fault vertices, unlike fault faces, must be
  - combined into faces on a fault mesh, and
  - oriented
- The fault mesh is used to
  - split vertices along the fault
  - introduce prism elements between adjacent fault faces
- Sieve code works for
  - any dimension
  - any element shape

Splitting the Mesh

- In order to create a fault, the generator provides
  - a set of fault vertices, or
  - a set of fault faces.
- Fault vertices, unlike fault faces, must be
  - combined into faces on a fault mesh, and
  - oriented
- The fault mesh is used to
  - split vertices along the fault
  - introduce prism elements between adjacent fault faces
- Sieve code works for
  - any dimension
  - any element shape

Splitting the Mesh

- In order to create a fault, the generator provides
  - a set of fault vertices, or
  - a set of fault faces.
- Fault vertices, unlike fault faces, must be
  - combined into faces on a fault mesh, and
  - oriented
- The fault mesh is used to
  - split vertices along the fault
  - introduce prism elements between adjacent fault faces
- Sieve code works for
  - any dimension
  - any element shape

Splitting the Mesh

- In order to create a fault, the generator provides
  - a set of fault vertices, or
  - a set of fault faces.
- Fault vertices, unlike fault faces, must be
  - combined into faces on a fault mesh, and
  - oriented
- The fault mesh is used to
  - split vertices along the fault
  - introduce prism elements between adjacent fault faces
- Sieve code works for
  - any dimension
  - any element shape

Splitting the Mesh

- In order to create a fault, the generator provides
  - a set of fault vertices, or
  - a set of fault faces.
- Fault vertices, unlike fault faces, must be
  - combined into faces on a fault mesh, and
  - oriented
- The fault mesh is used to
  - split vertices along the fault
  - introduce prism elements between adjacent fault faces
- Sieve code works for
  - any dimension
  - any element shape

Outline