Implementation for Scientific Computing: FEM and FMM

Matthew Knepley

Computation Institute
University of Chicago

Department of Mathematics
Tufts University
March 12, 2010

Outline

Computational Mathematics

can produce Better Software

and lead to Better Science

Computational Mathematics can produce Better Software and lead to Better Science

Computational Mathematics can produce Better Software and lead to Better Science

How?

Improve Accuracy, Stability, or Scaling

- Spectral elements
- SUPG
- Multigrid

How?

Automatically Optimize

- Loop Tiling
- FErari
- PetFMM

How?

Simplify Design

- Generic type systems
- Sieve
- PetFMM

How?

Explore Algorithmic Tradeoffs

- Treecode vs. FMM
- Conforming vs. Nonconforming elements
- FMM vs. Multigrid for Poisson on a GPU

Collaborators

- Automated FEM
  - Andy Terrel (UT Austin)
  - Ridgway Scott (UChicago)
  - Rob Kirby (Texas Tech)
- Sieve
  - Dmitry Karpeev (ANL)
  - Peter Brune (UChicago)
  - Anders Logg (Simula)
- FMM
  - Lorena Barba (BU)
  - Felipe Cruz (Bristol)
  - Rio Yokota (BU)

Outline

Main Point

A familiar problem,
FEM assembly,

is recast to allow

automatic optimization.

Main Point

A familiar problem,

FEM assembly,

is recast to allow

automatic optimization.

Main Point

A familiar problem,

FEM assembly,

is recast to allow

automatic optimization.

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.

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

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

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. (6*b*) : Basis expansion

Eq. (6*c*) : Exclude nonbasis vector from the expansion

Eq. (6*d*) : Remove offdiagonal coefficients for basis vectors

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

Original Formulation

MINLP Model

$$\text{minimize} \quad \sum_{i=1}^{n} \left\{ y_i(2m-1) + (1-y_i)\left(2\sum_{j=1, j\neq i}^{n} y_j - 1\right)\right\} \tag{6a}$$

$$\text{subject to} \quad v_i = \sum_{j=1}^{n} \alpha_{ij} v_j \qquad i = 1, \ldots, n \tag{6b}$$

$$-Uy_j \leq \alpha_{ij} \leq Uy_j \qquad i, j = 1, \ldots, n \tag{6c}$$

$$-U(1-y_i) \leq \alpha_{ij} \leq U(1-y_i) \qquad i, j = 1, \ldots, n, \tag{6d}$$

$$y_i \in \{0, 1\} \qquad i = 1, \ldots, n. \tag{6e}$$

Original Formulation

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

$$\text{minimize} \quad 2m\sum_{i=1}^{n} y_i + 2\sum_{i=1}^{n}\sum_{j=1, j\neq i}^{n} (y_j - z_{ij}) - n \tag{6a}$$

$$\text{subject to} \quad v_i = \sum_{j=1}^{n} \alpha_{ij} v_j \qquad i = 1, \ldots, n \tag{6b}$$

$$-Uy_j \leq \alpha_{ij} \leq Uy_j \qquad i, j = 1, \ldots, n \tag{6c}$$

$$-U(1 - y_i) \leq \alpha_{ij} \leq U(1 - y_i) \qquad i, j = 1, \ldots, n,\ i \neq j \tag{6d}$$

$$z_{ij} \leq y_i, \quad z_{ij} \leq y_j, \quad z_{ij} \geq y_i + y_j - 1, \qquad i, j = 1, \ldots, n \tag{6e}$$

$$y_i \in \{0, 1\}, \quad z_{ij} \in \{0, 1\} \qquad i, j = 1, \ldots, n. \tag{6f}$$

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

$$\text{minimize} \quad \sum_{i=1}^{n} \left\{ y_i(2m-1) + (1-y_i)\left(2\sum_{j=1, j\neq i}^{n} s_{ij} - 1\right)\right\} \tag{7a}$$

$$\text{subject to} \quad v_i = \sum_{j=1}^{n} \alpha_{ij} v_j \qquad i = 1, \ldots, n \tag{7b}$$

$$-Us_{ij} \leq \alpha_{ij} \leq Us_{ij} \qquad i, j = 1, \ldots, n \tag{7c}$$

$$-U(1-y_i) \leq \alpha_{ij} \leq U(1-y_i) \qquad i, j = 1, \ldots, n \tag{7d}$$

$$s_{ij} \leq y_j \qquad i, j = 1, \ldots, n \tag{7e}$$

$$y_i \in \{0,1\}, \quad s_{ij} \in \{0,1\} \qquad i, j = 1, \ldots, n \tag{7f}$$

Sparse Coefficient Formulation

Equivalent MILP Model

$$\text{minimize} \quad 2m\sum_{i=1}^{n} y_i + 2\sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n}(s_{ij} - z_{ij}) - n \tag{7a}$$

$$\text{subject to} \quad v_i = \sum_{j=1}^{n} \alpha_{ij} v_j \qquad i = 1, \ldots, n \tag{7b}$$

$$-Us_{ij} \leq \alpha_{ij} \leq Us_{ij} \qquad i, j = 1, \ldots, n \tag{7c}$$

$$-U(1 - y_i) \leq \alpha_{ij} \leq U(1 - y_i) \qquad i, j = 1, \ldots, n,\ i \neq \tag{7d}$$

$$z_{ij} \leq y_i, \quad z_{ij} \leq s_{ij}, \quad z_{ij} \geq y_i + s_{ij} - 1, \qquad i, j = 1, \ldots, n \tag{7e}$$

$$y_i \in \{0, 1\}, \quad z_{ij} \in \{0, 1\}, \quad s_{ij} \in \{0, 1\} \qquad i, j = 1, \ldots, n. \tag{7f}$$

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

Main Point

Rethinking meshes

produces a simple FEM interface

and good code reuse.

Main Point

Rethinking meshes

produces a simple FEM interface

and good code reuse.

Main Point

Rethinking meshes

produces a simple FEM interface

and good code reuse.

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?"

Outline

Sieve

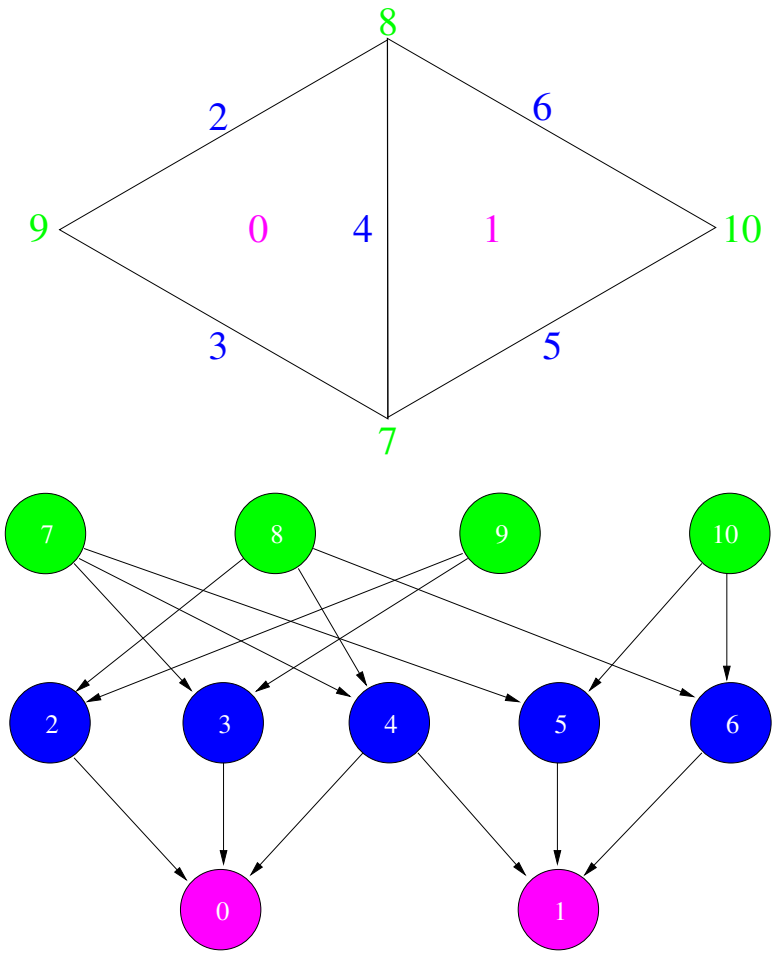
Sieve is an interface for

- general topologies
- functions over these topologies (bundles)
- traversals

One relation handles all hierarchy

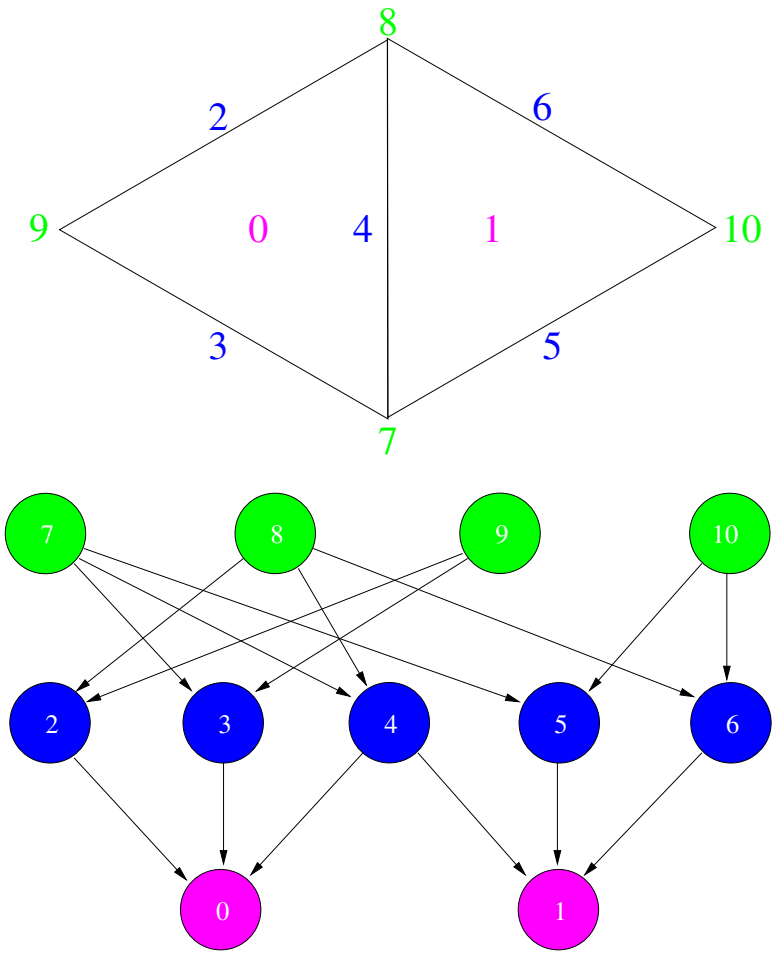
- Vast reduction in complexity
  - Dimension independent code
  - A single communication routine to optimize
- Expansion of capabilities
  - Partitioning and distribution
  - Hybrid meshes
  - Complicated structures and embedded boundaries
  - Unstructured multigrid

Doublet Mesh



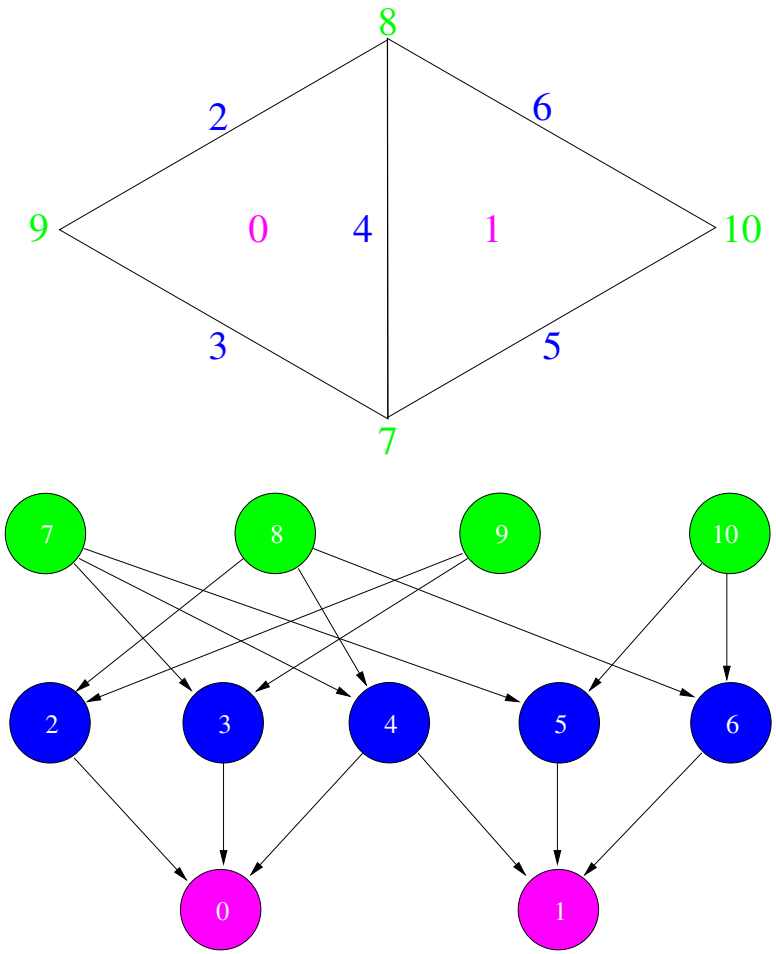
- Incidence/covering arrows
- $cone(0) = \{2, 3, 4\}$
- $support(7) = \{2, 3\}$

Doublet Mesh



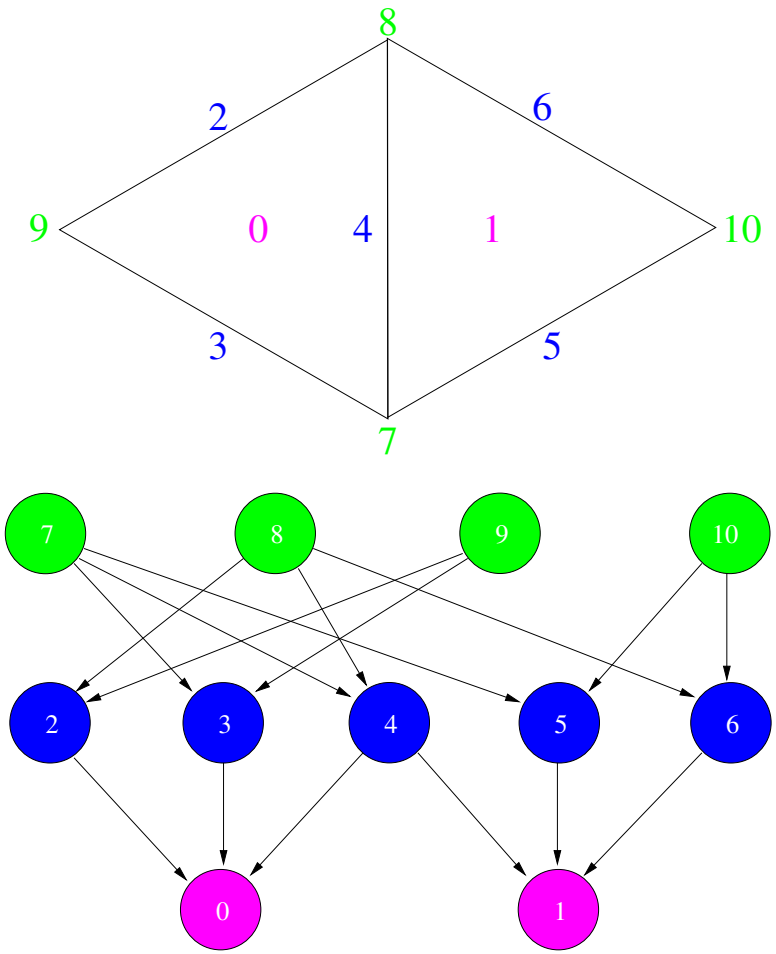
- Incidence/covering arrows
- $cone(0) = \{2, 3, 4\}$
- $support(7) = \{2, 3\}$

Doublet Mesh



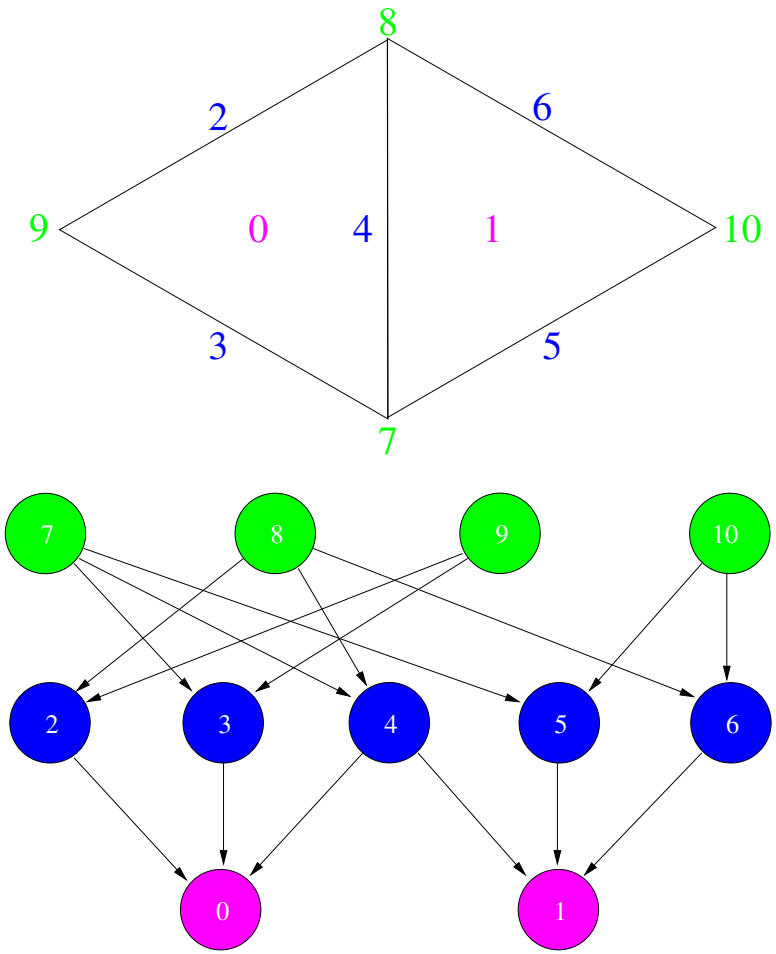
- Incidence/covering arrows
- $cone(0) = \{2, 3, 4\}$
- $support(7) = \{2, 3\}$

Doublet Mesh



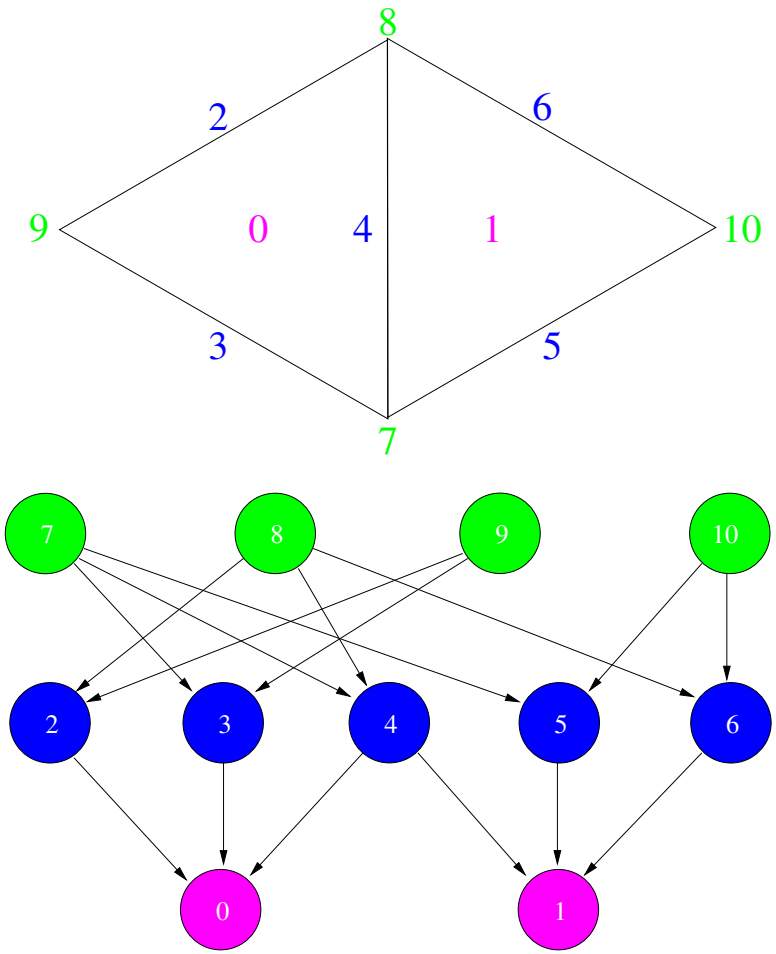
- Incidence/covering arrows
- $closure(0) = \{0, 2, 3, 4, 7, 8, 9\}$
- $star(7) = \{7, 2, 3, 0\}$

Doublet Mesh



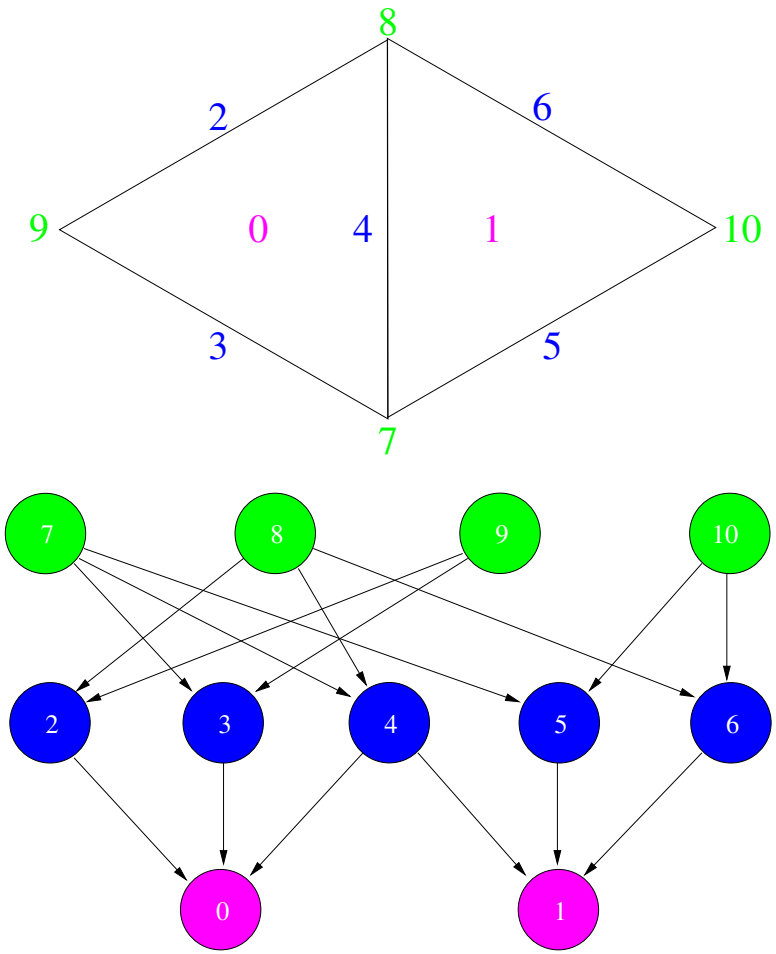
- Incidence/covering arrows
- $closure(0) = \{0, 2, 3, 4, 7, 8, 9\}$
- $star(7) = \{7, 2, 3, 0\}$

Doublet Mesh



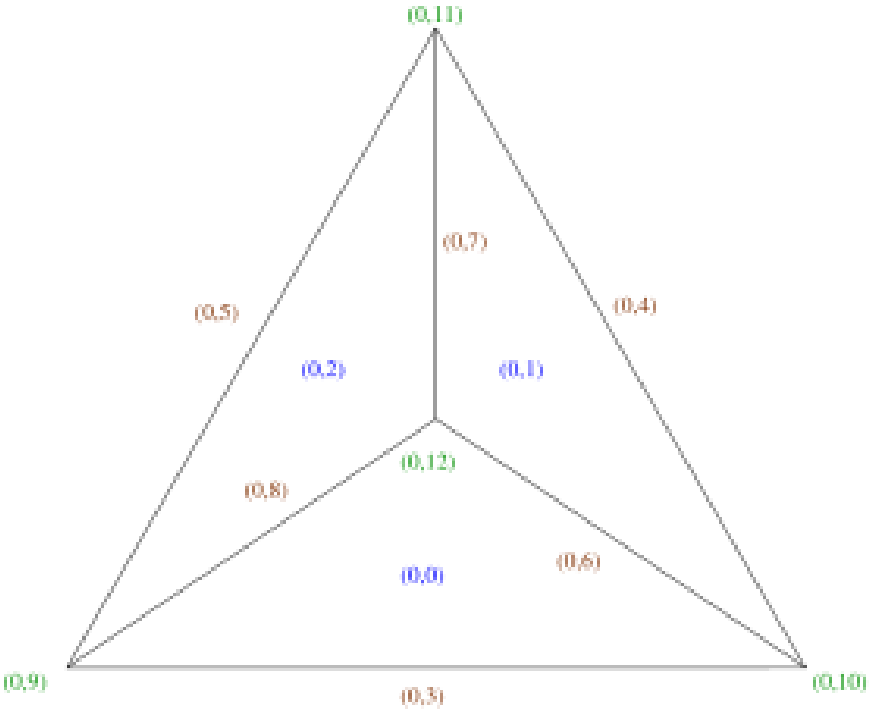
- Incidence/covering arrows
- $meet(0, 1) = \{4\}$
- $join(8, 9) = \{4\}$

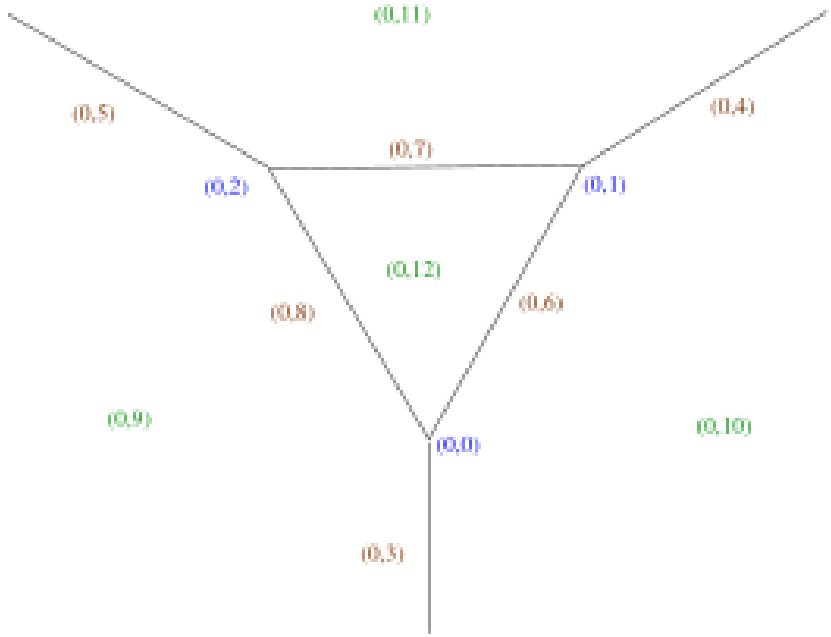
Doublet Mesh

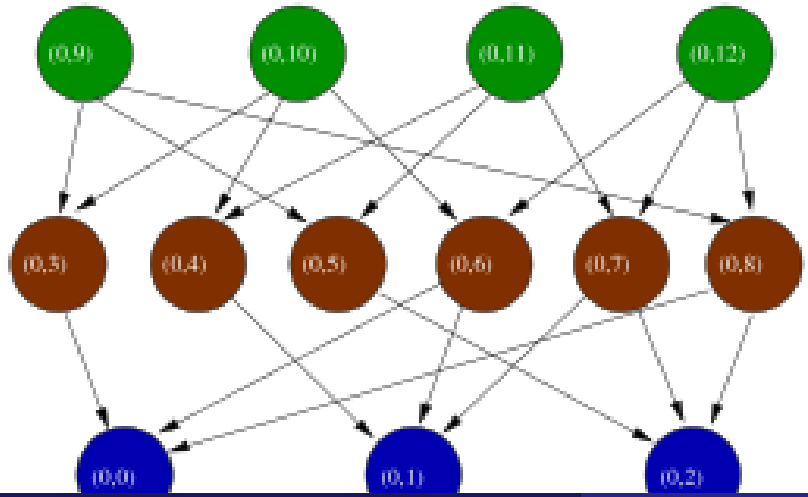


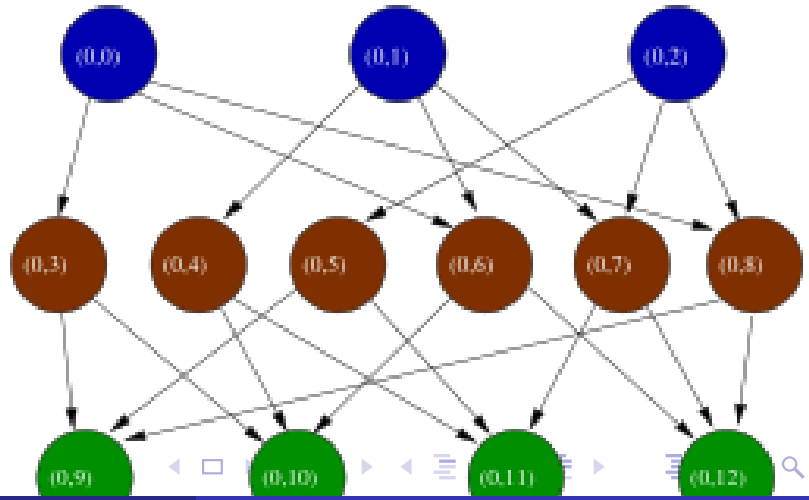
- Incidence/covering arrows
- $meet(0, 1) = \{4\}$
- $join(8, 9) = \{4\}$

The Mesh Dual

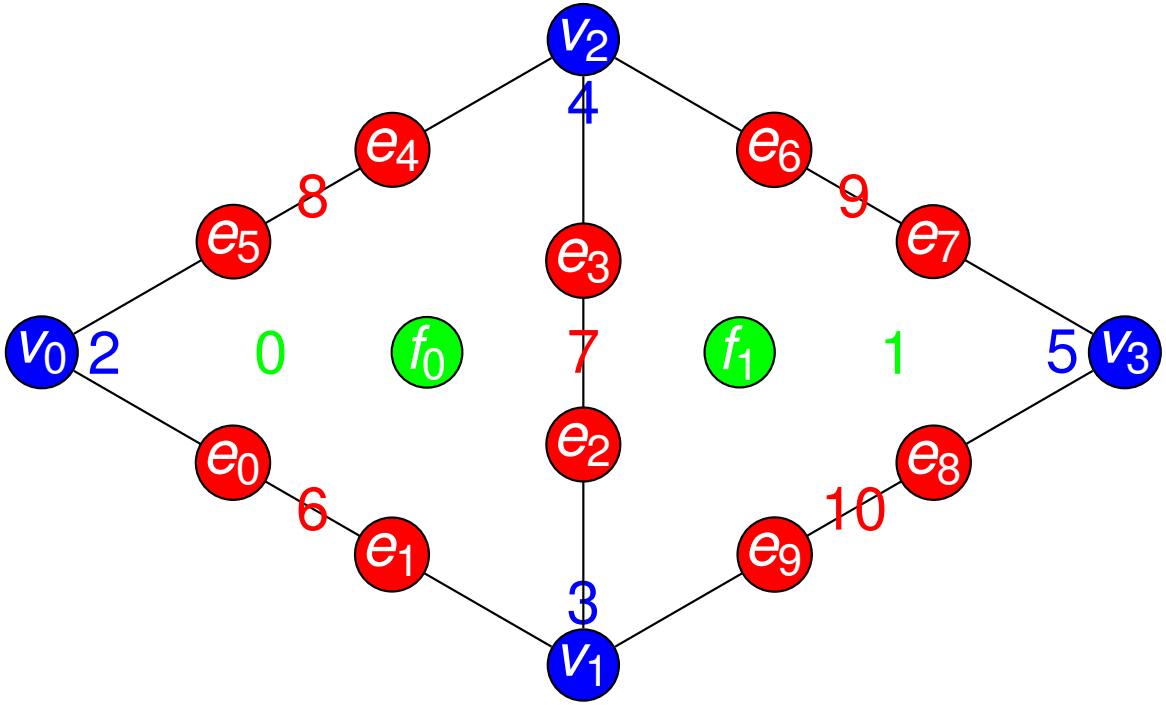


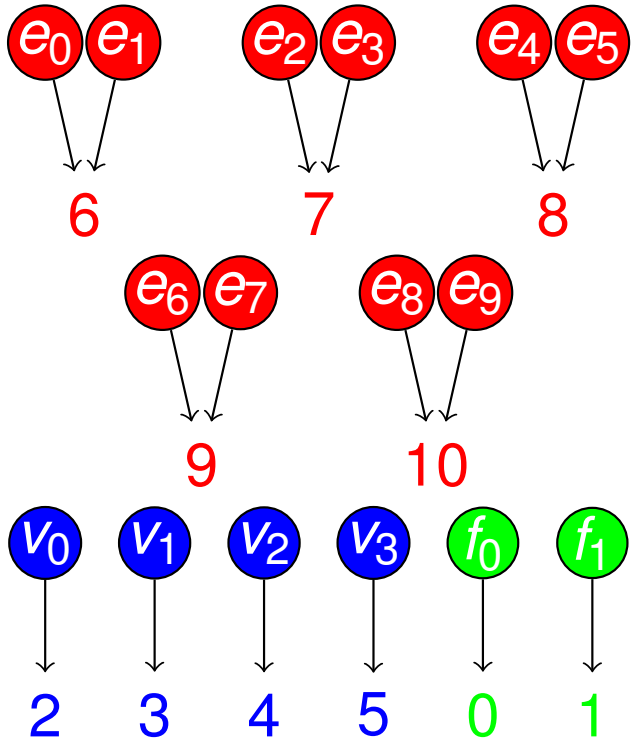






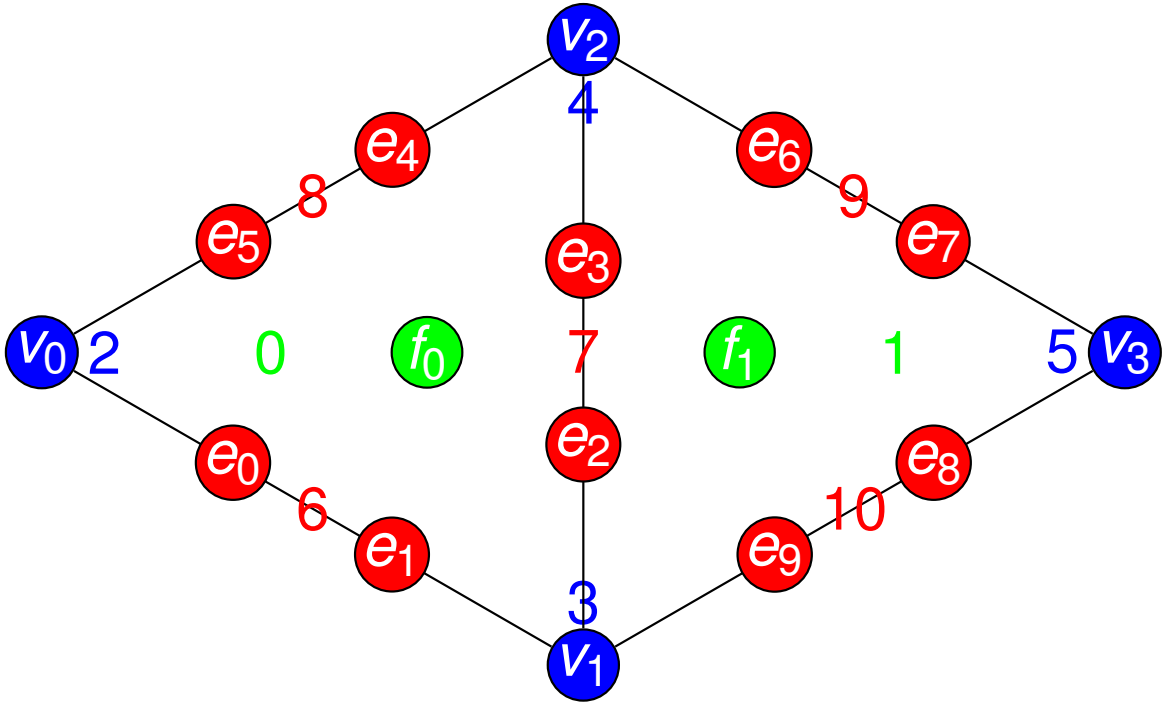
Doublet Section

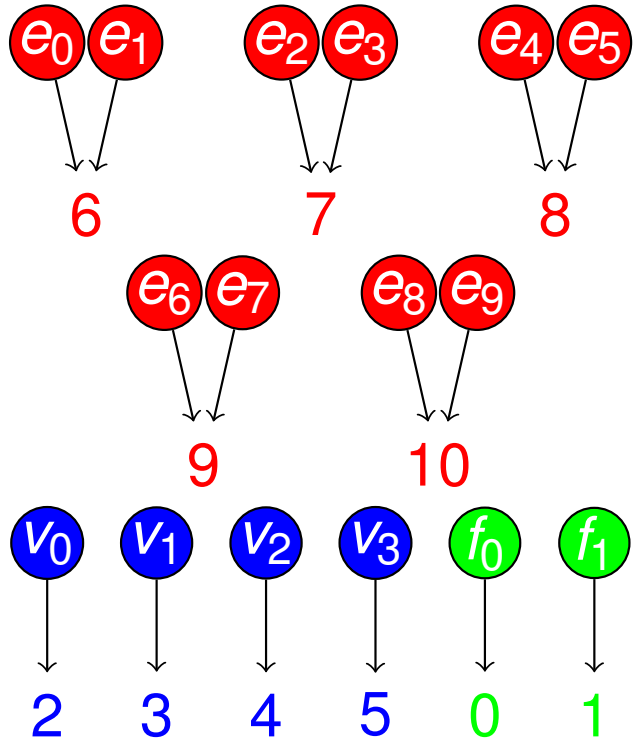




- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

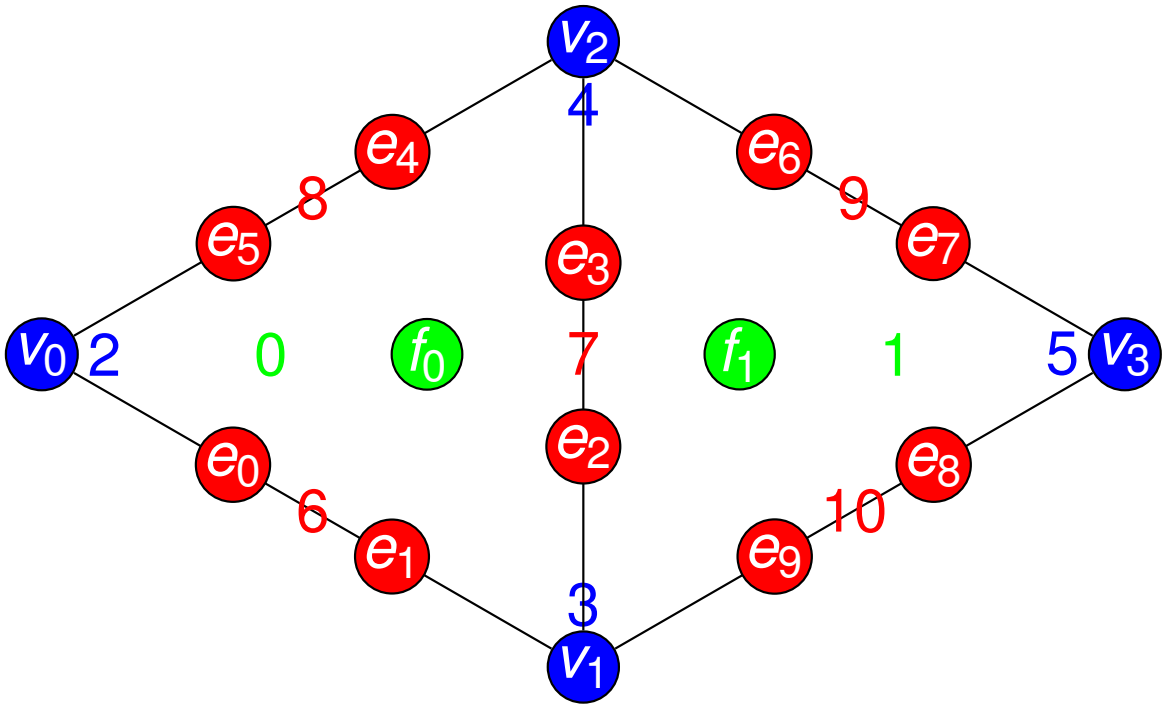
Doublet Section

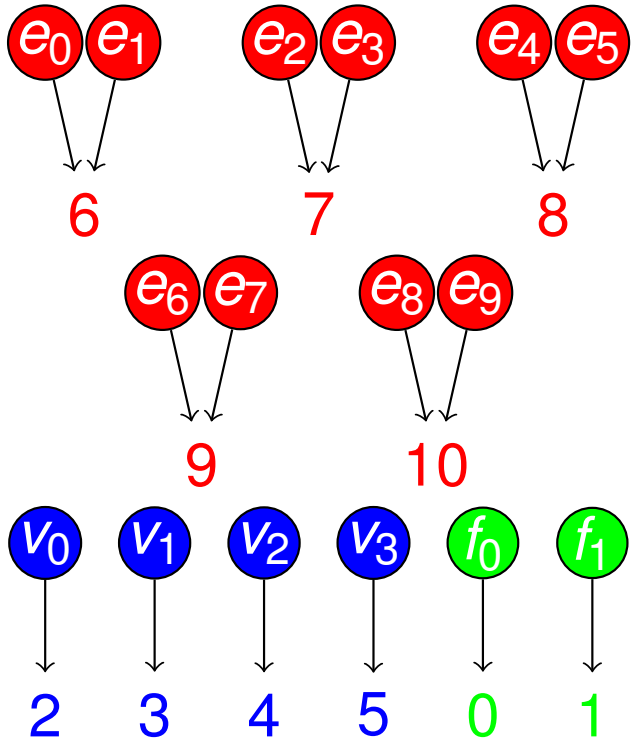




- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

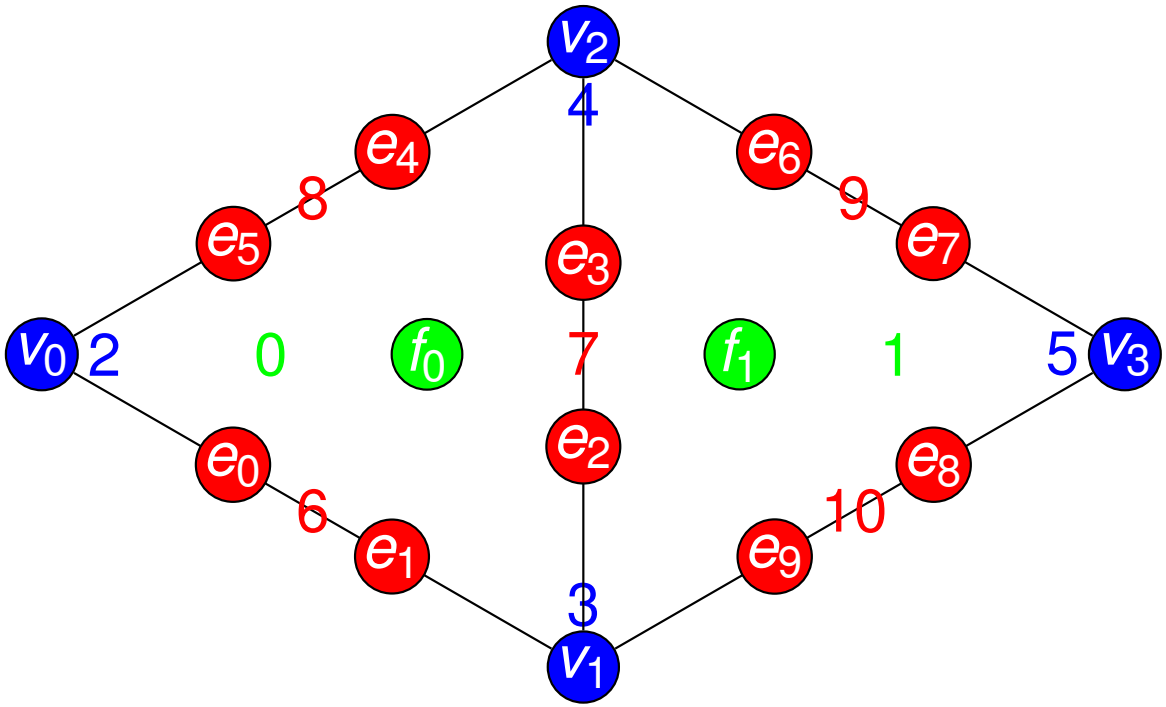
Doublet Section

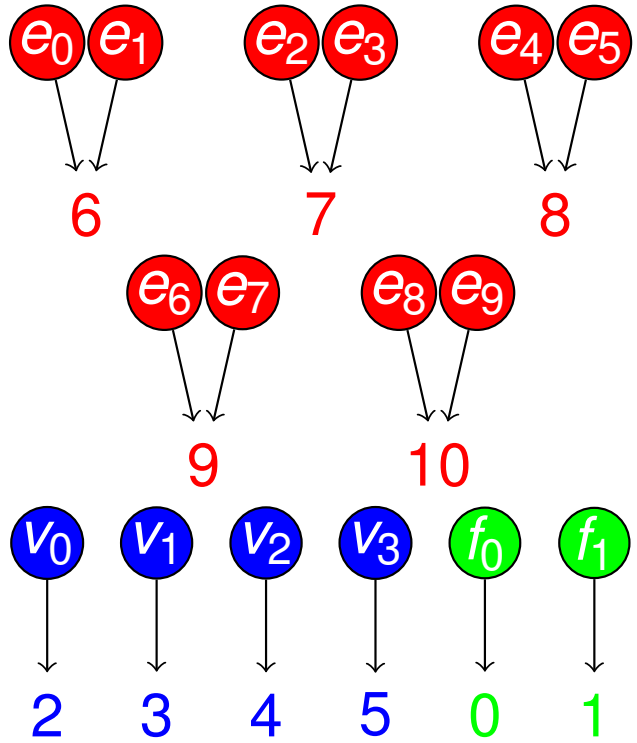




- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

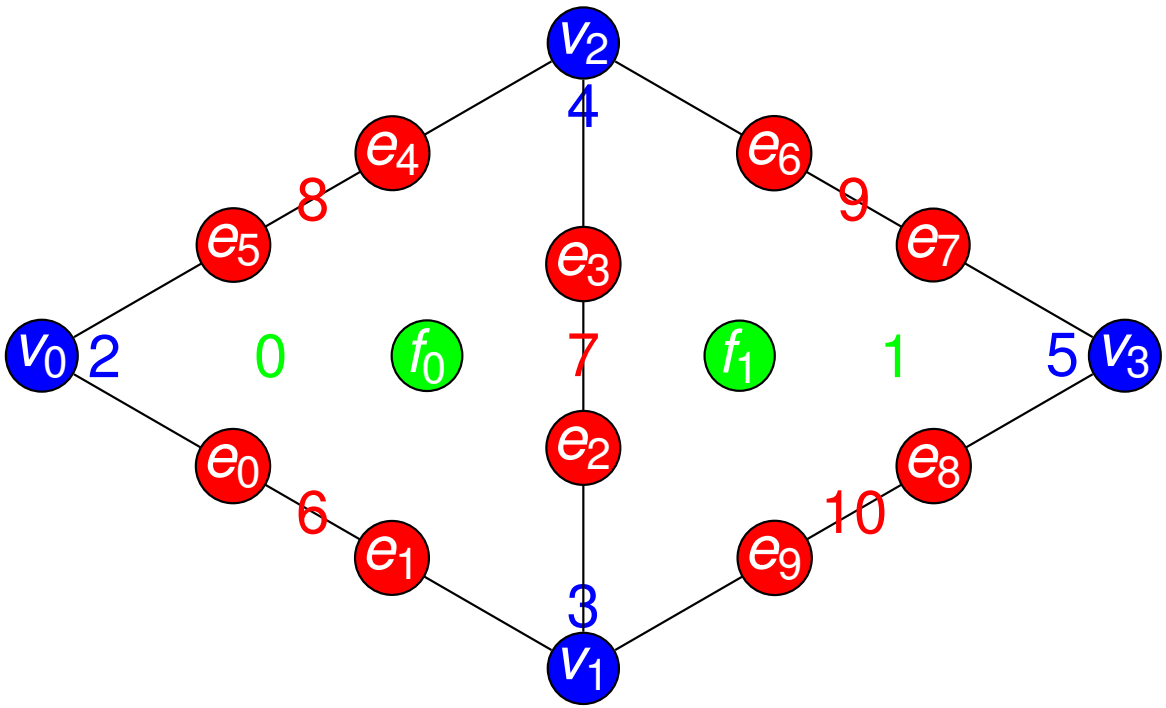
Doublet Section

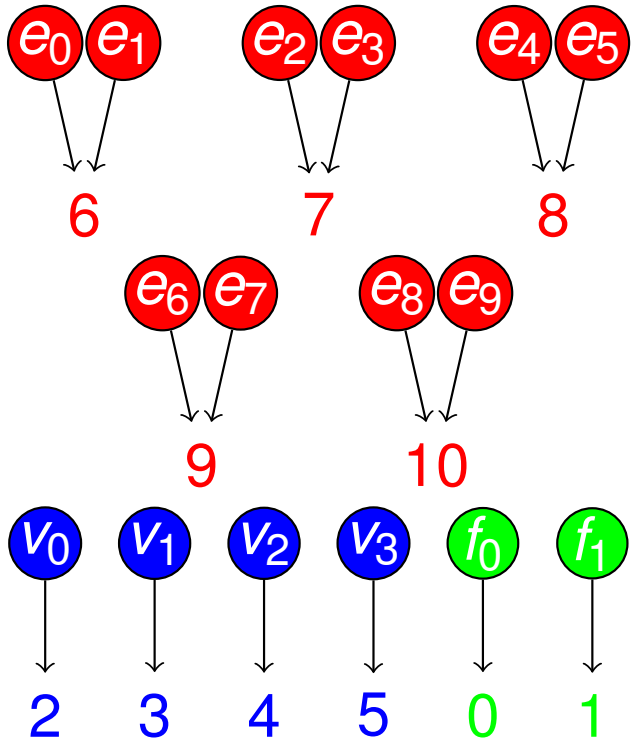




- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

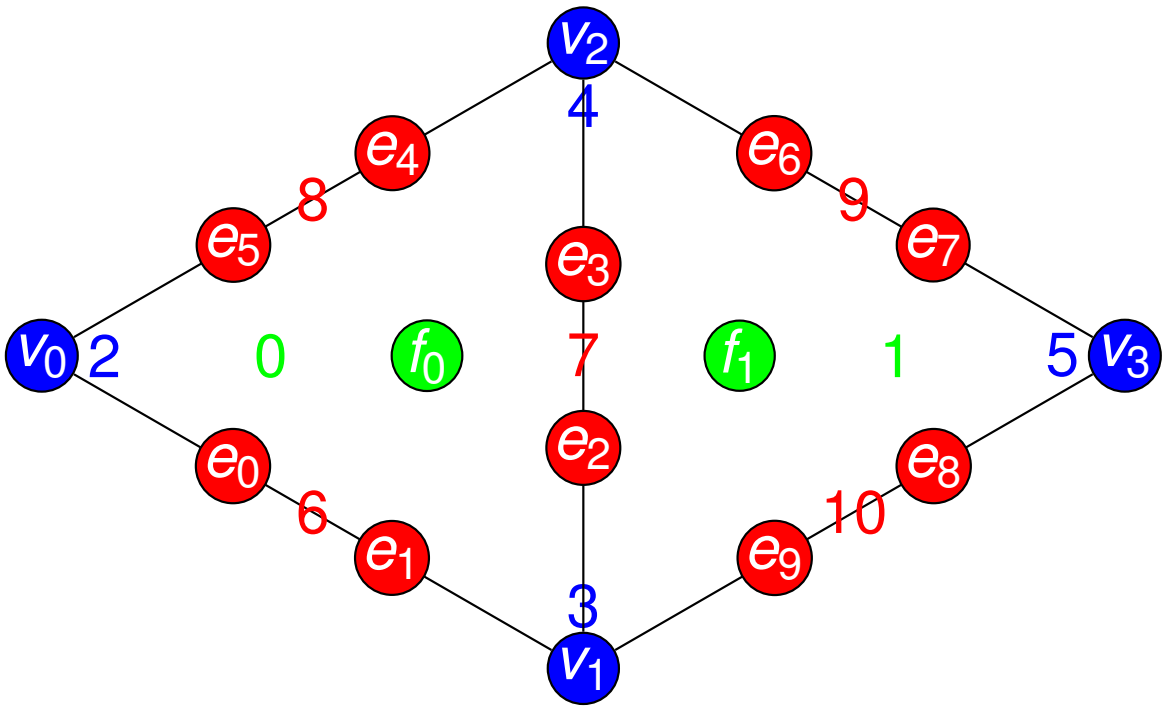
Doublet Section

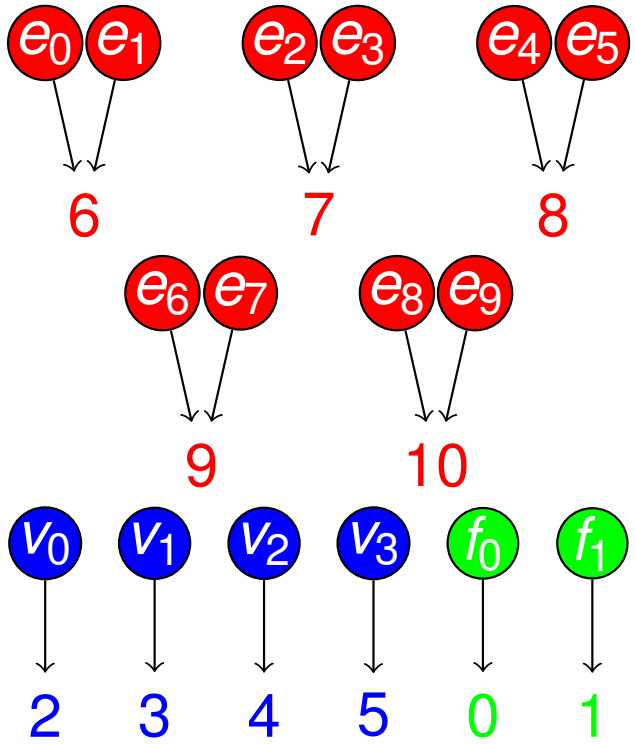




- Topological traversals: follow connectivity
  - $\mathit{restrictClosure}(0) = \{f_0\, e_0\, e_1\, e_2\, e_3\, e_4\, e_5\, v_0\, v_1\, v_2\}$
  - $\mathit{restrictStar}(7) = \{v_0\, e_0\, e_1\, e_4\, e_5\, f_0\}$

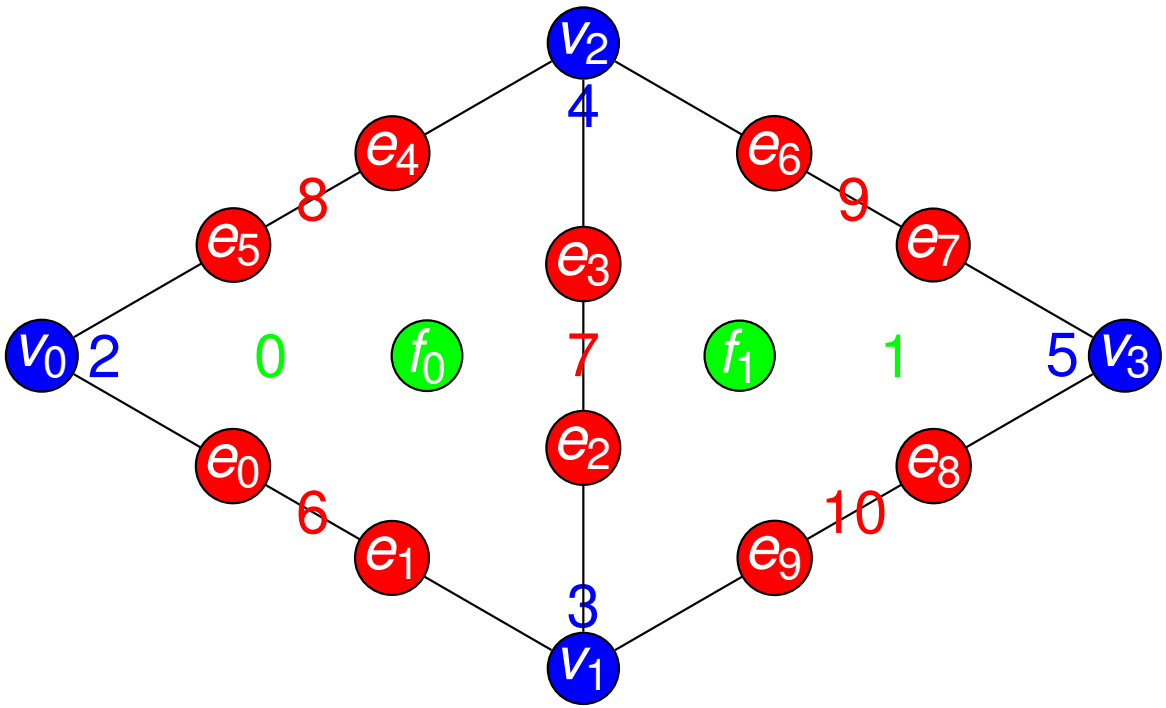
Doublet Section

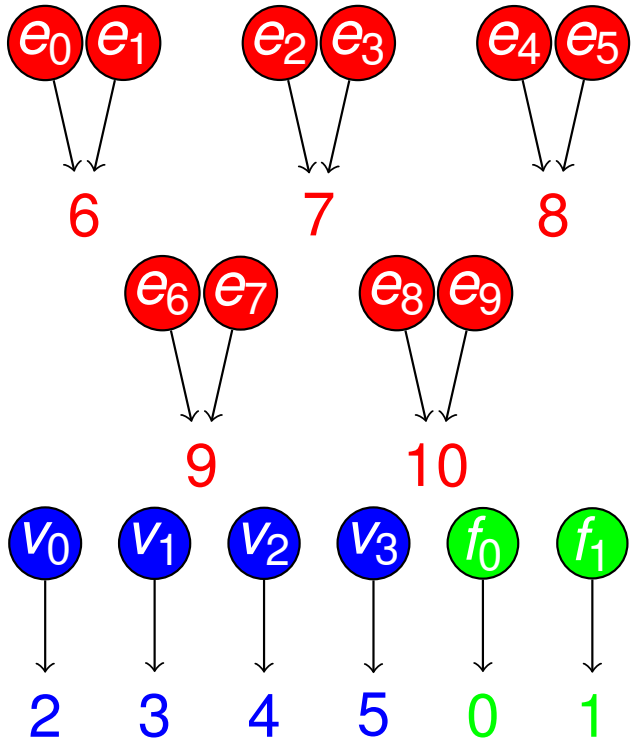




- Topological traversals: follow connectivity
  - *restrictClosure*(0) = $\{f_0\, e_0\, e_1\, e_2\, e_3\, e_4\, e_5\, v_0\, v_1\, v_2\}$
  - *restrictStar*(7) = $\{v_0\, e_0\, e_1\, e_4\, e_5\, f_0\}$

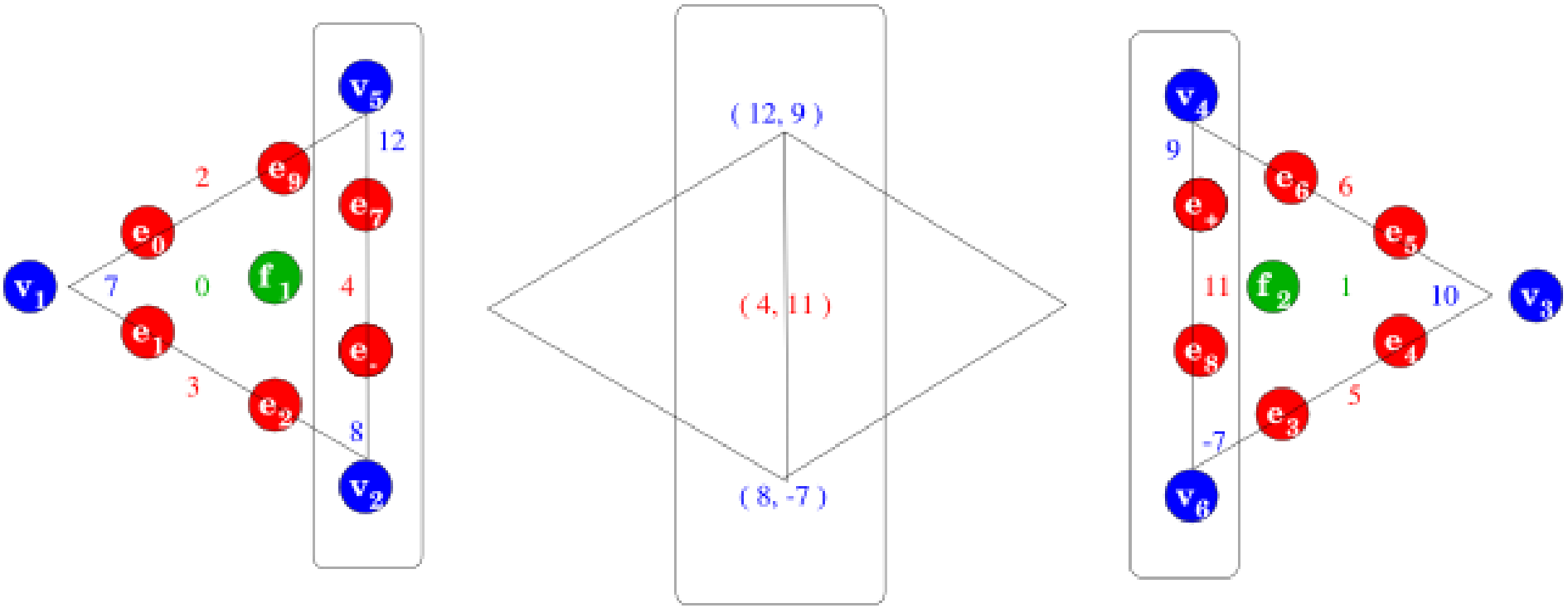
Doublet Section





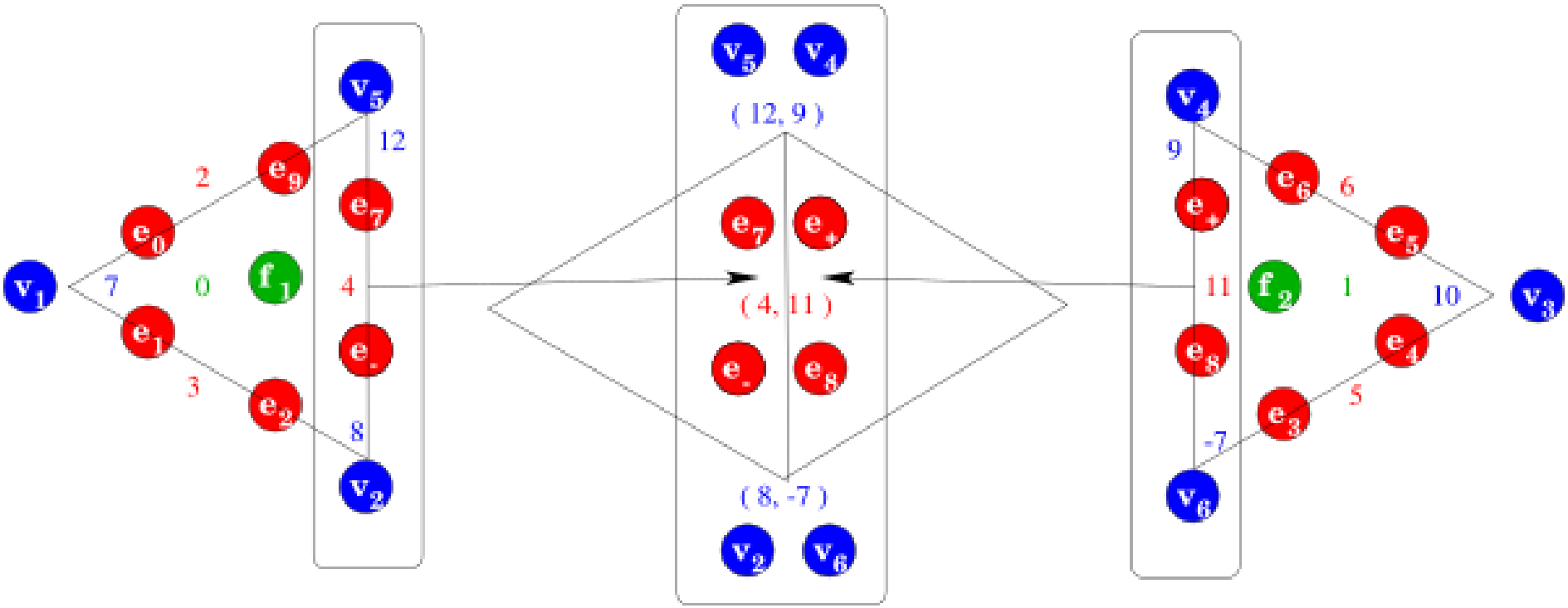
- Topological traversals: follow connectivity
  - $\mathit{restrictClosure}(0) = \{f_0\, e_0\, e_1\, e_2\, e_3\, e_4\, e_5\, v_0\, v_1\, v_2\}$
  - $\mathit{restrictStar}(7) = \{v_0\, e_0\, e_1\, e_4\, e_5\, f_0\}$

Restriction



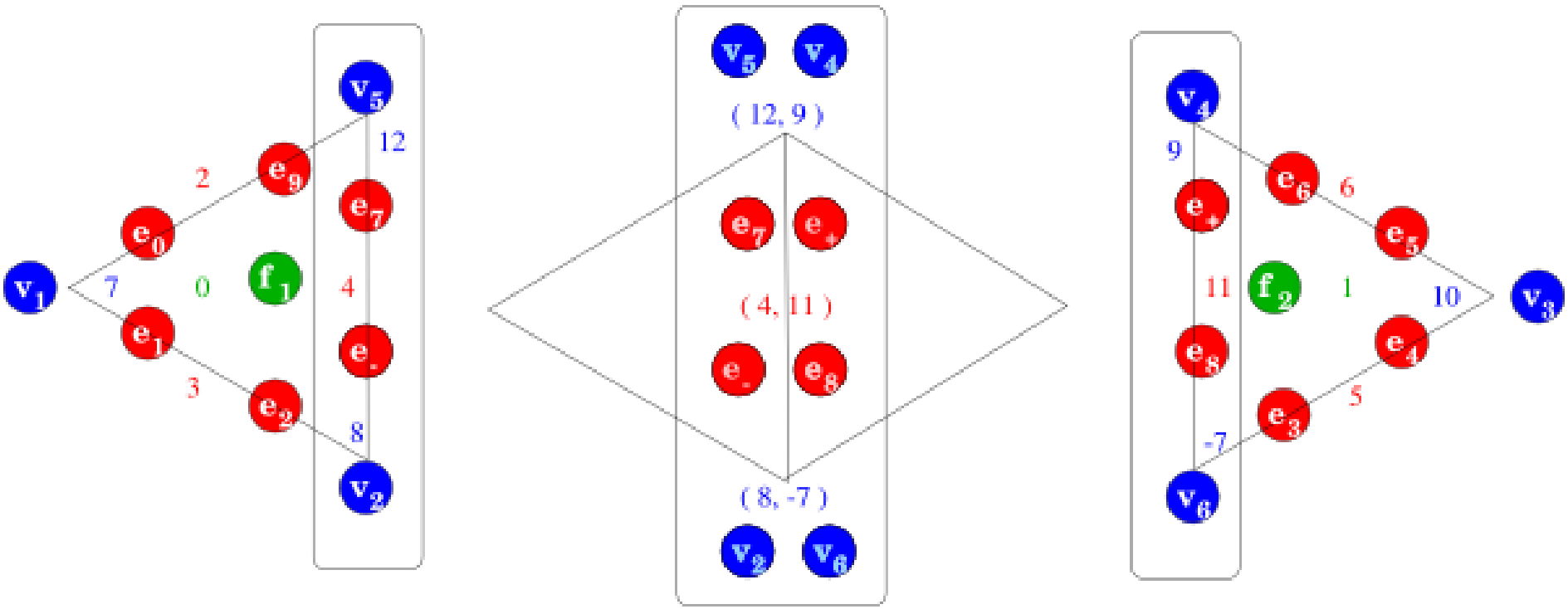
- Localization
  - Restrict to patches (here an edge closure)
  - Compute locally

Delta



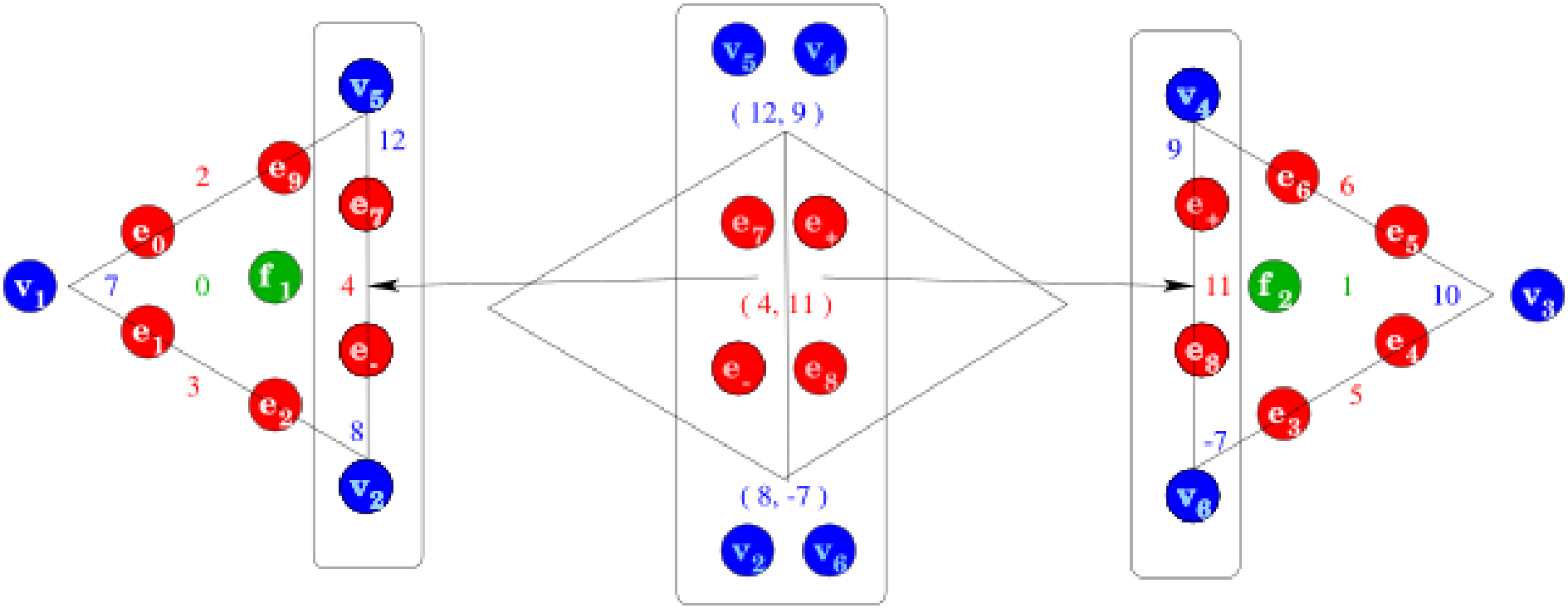
- `Delta`
  - Restrict further to the overlap
  - `Overlap` now carries twice the data

Fusion



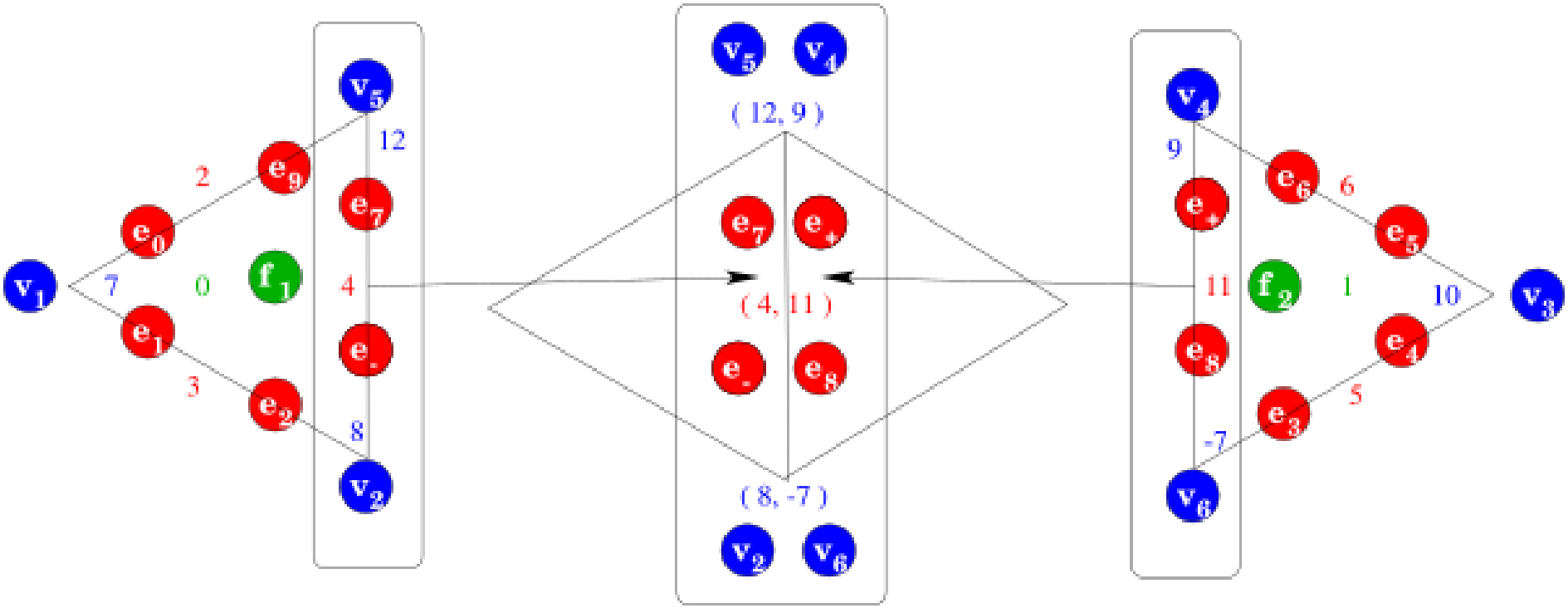
- Merge/reconcile data on the overlap
  - Addition (FEM)
  - Replacement (FD)
  - Coordinate transform (Sphere)
  - Linear transform (MG)

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

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

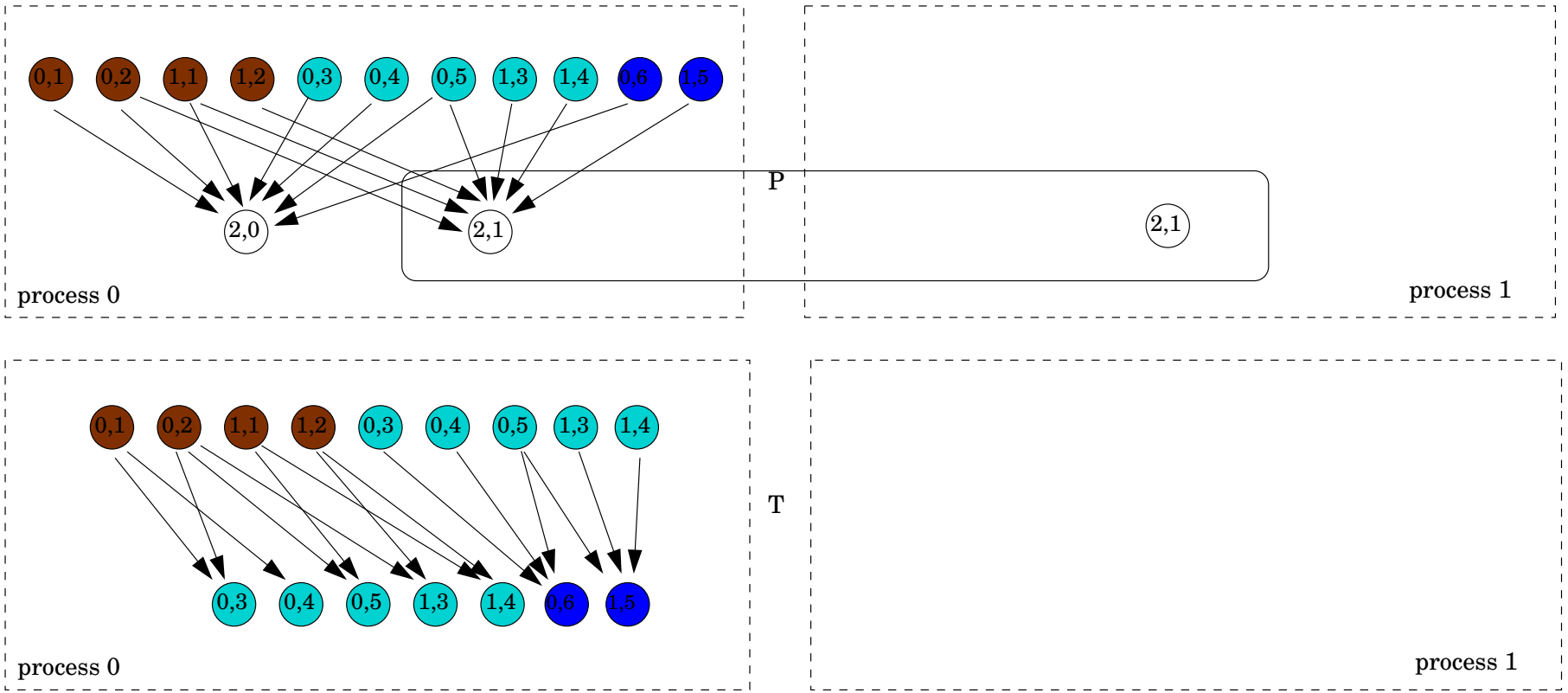
- distributing the topology (Sieve)
- distributing data (Section)

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

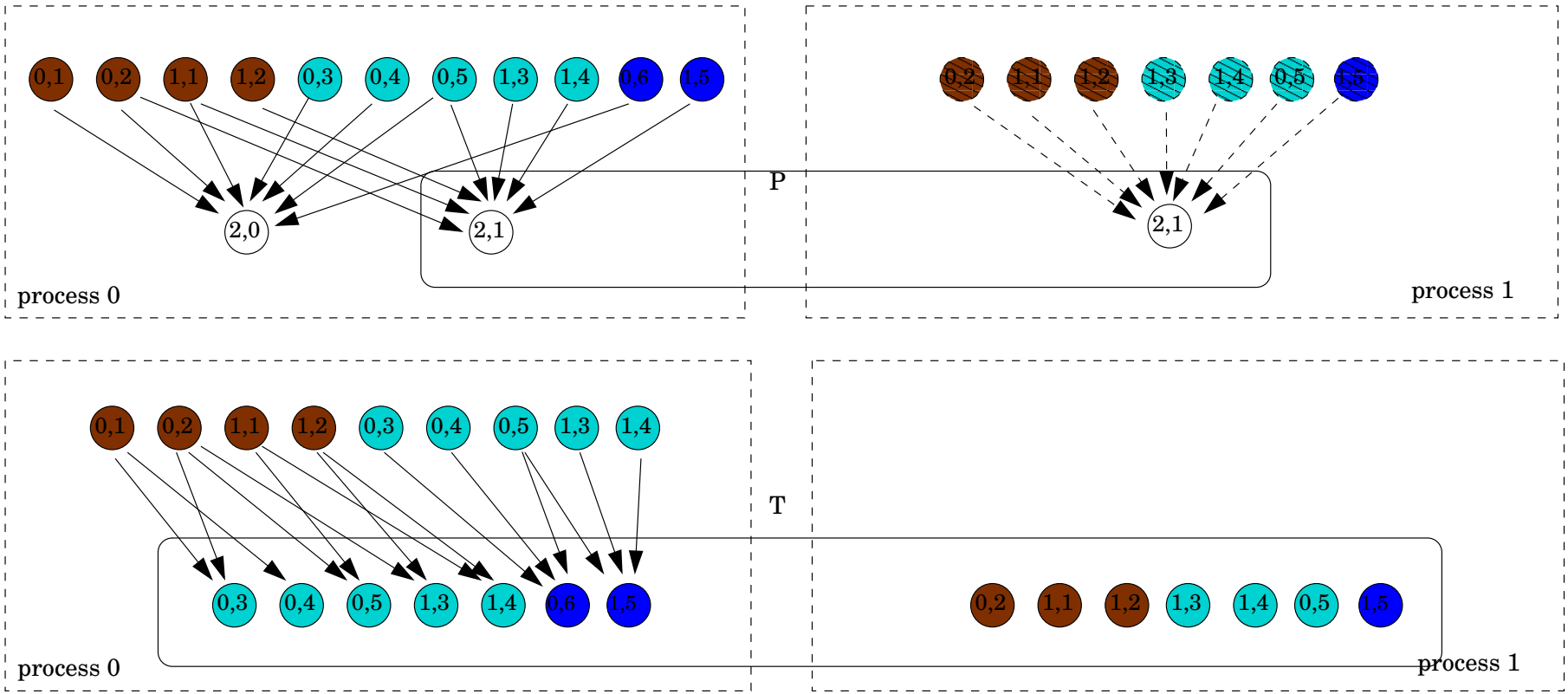
Mesh Partition

- 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

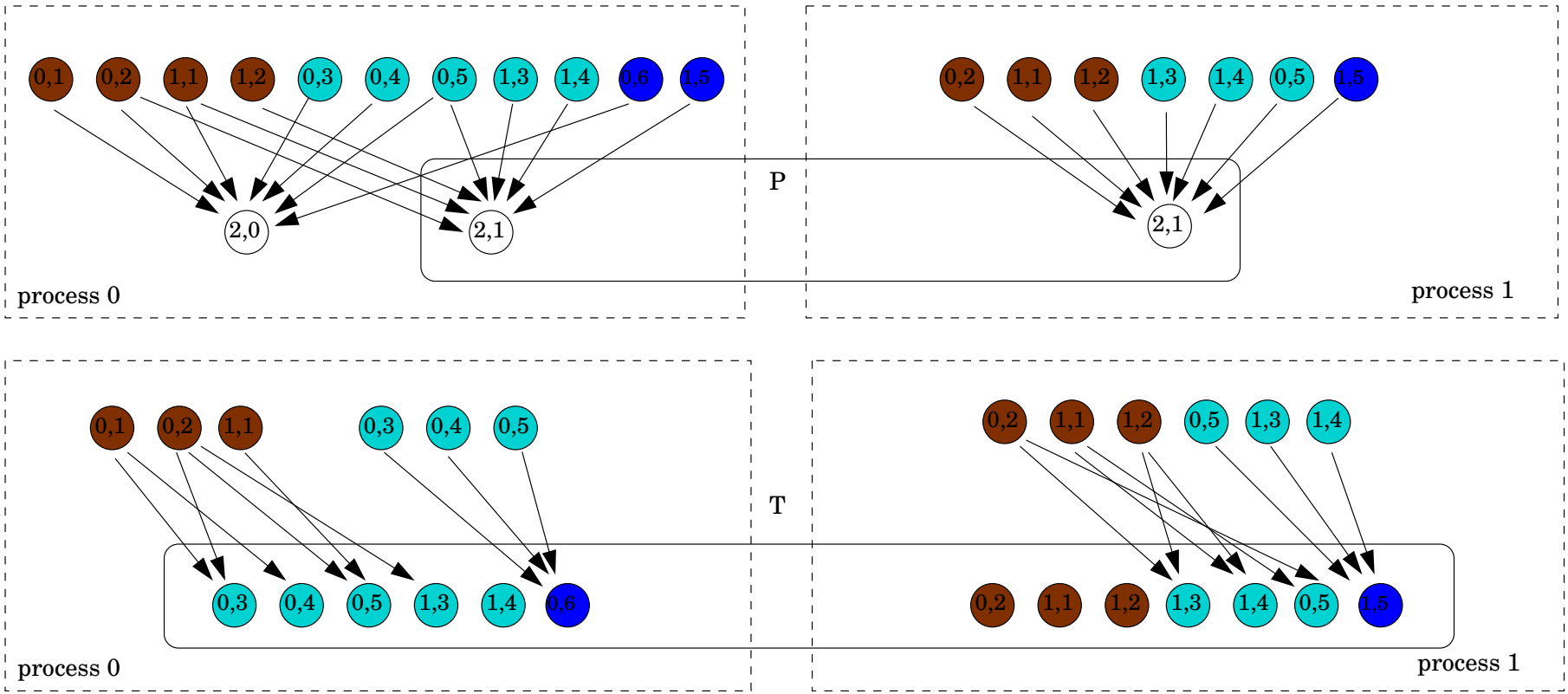
Doublet Mesh Distribution



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

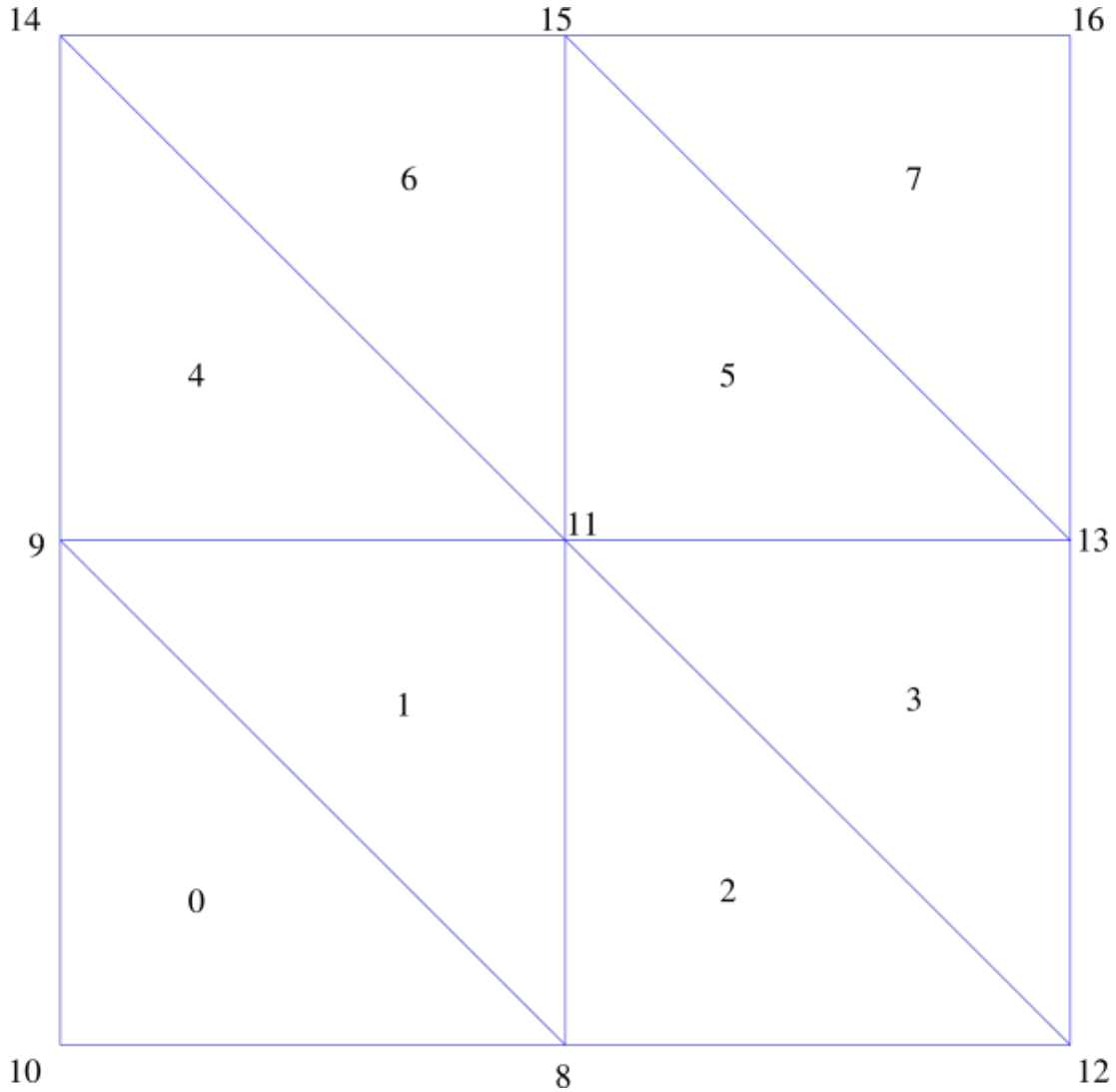
1. Construct local mesh from partition
2. Construct partition overlap
3. `Complete()` the partition section
   - This distributes the cells
4. Update Overlap with new points
5. `Complete()` the cone section
   - This distributes the remaining sieve points
6. Update local Sieves

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

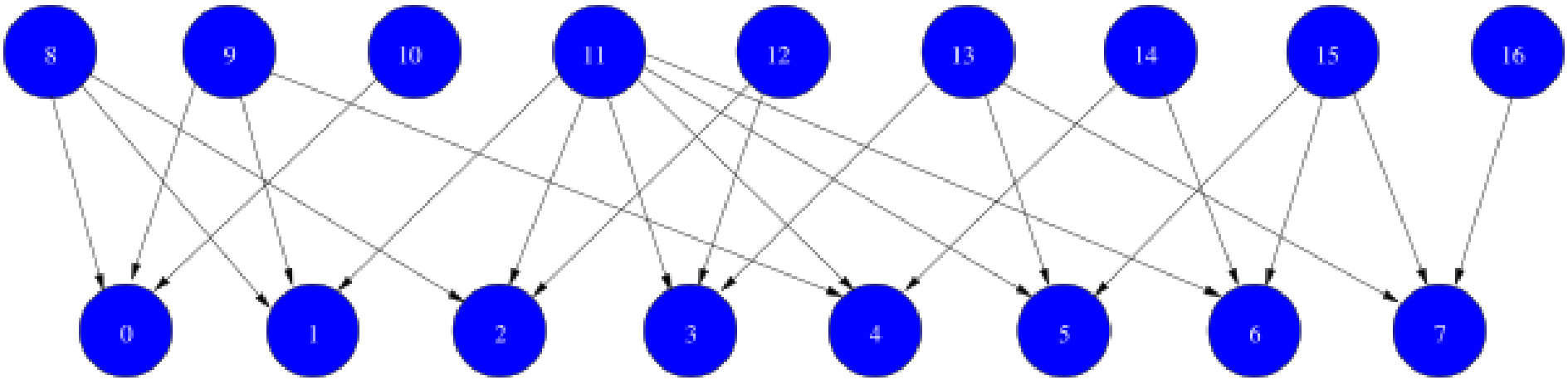
2D Example

A simple triangular mesh



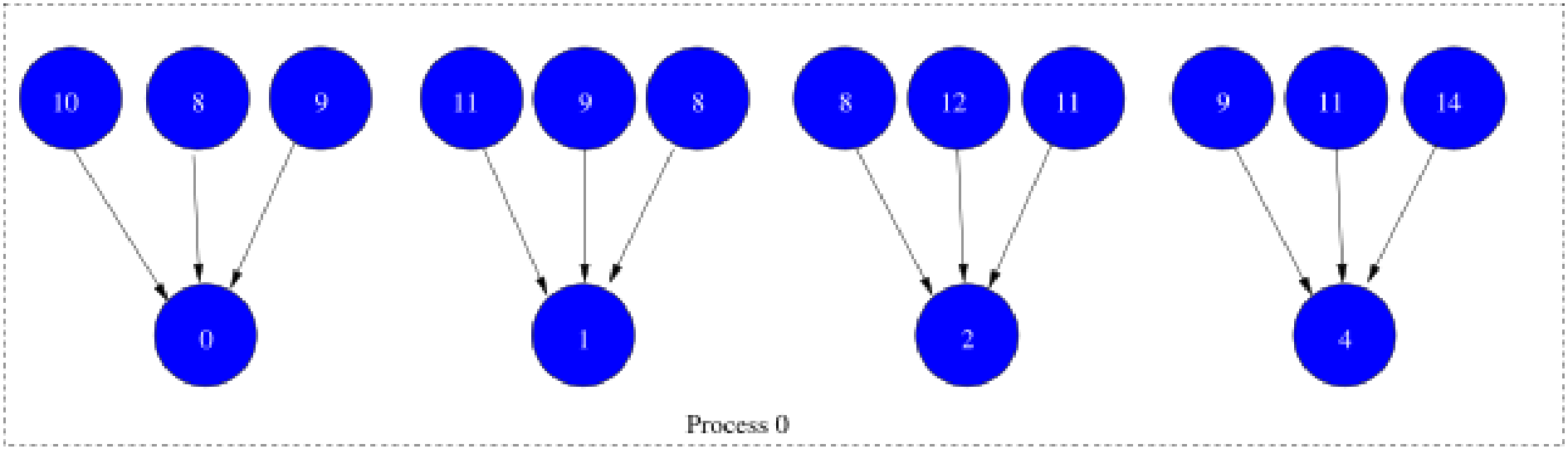
2D Example

Sieve for the mesh



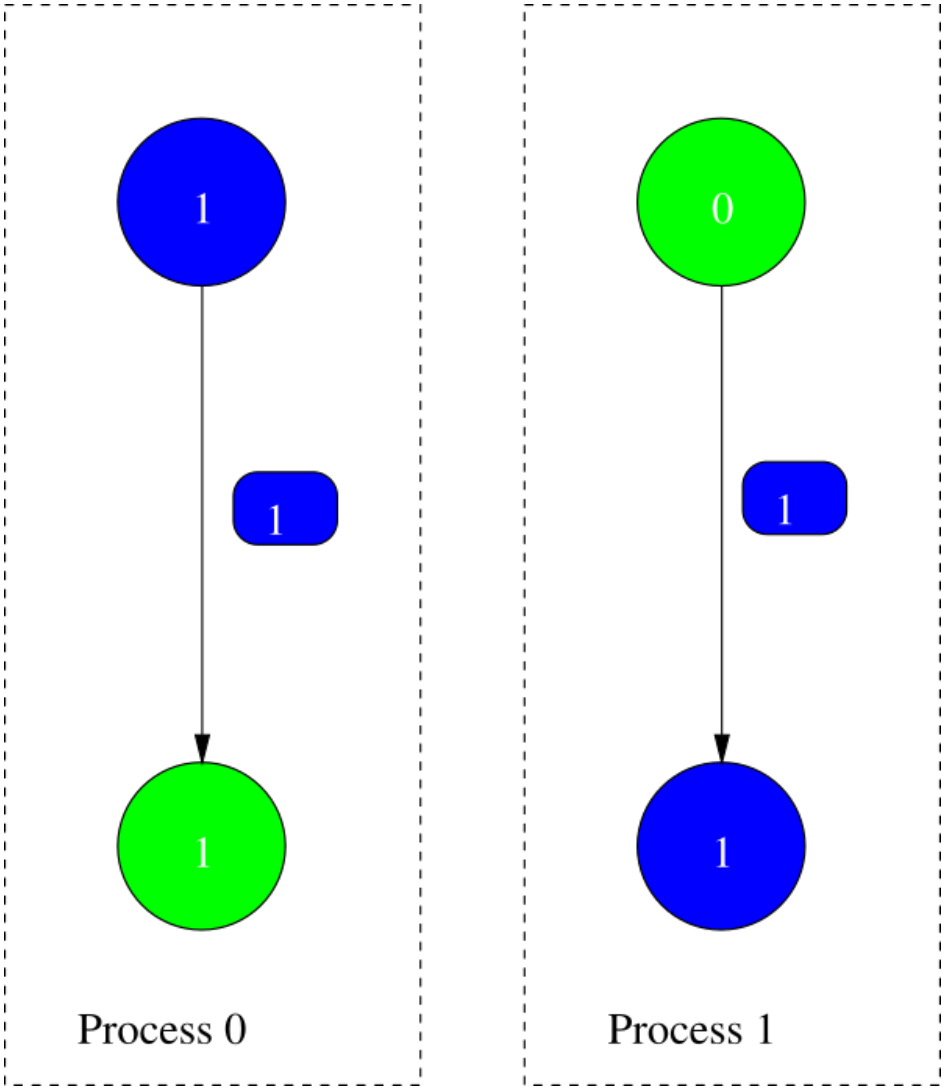
2D Example

Local sieve on process 0



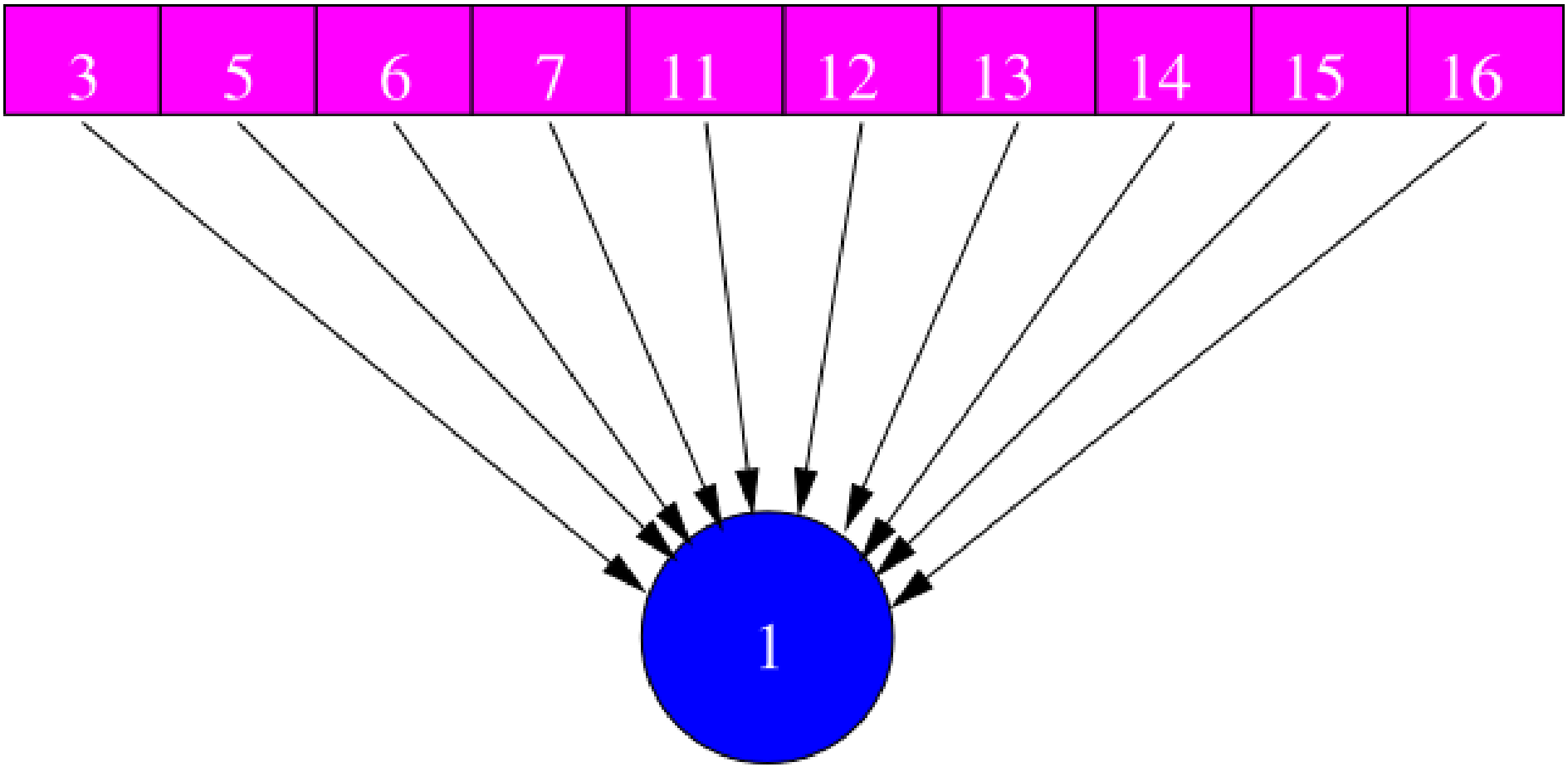
2D Example

Partition Overlap



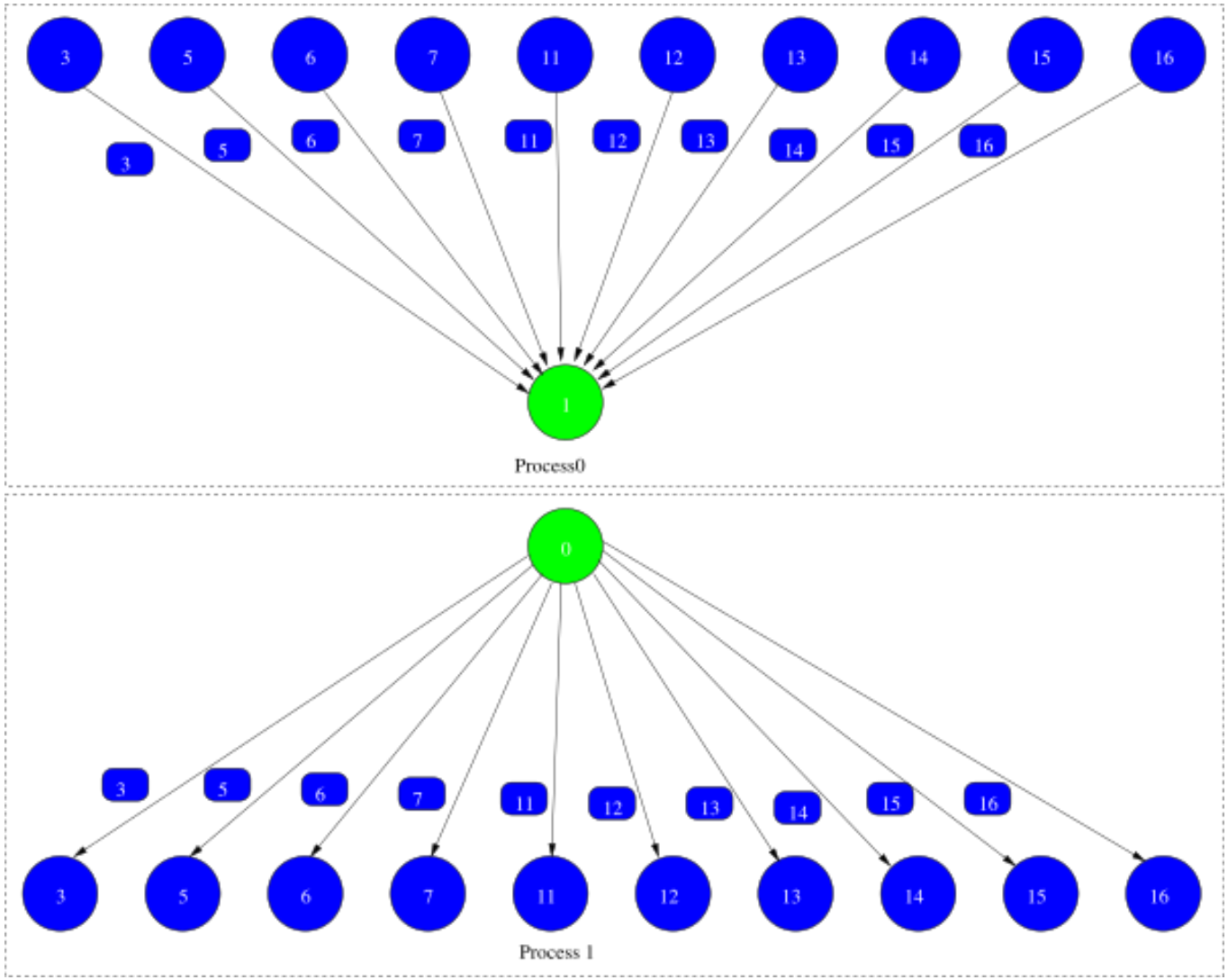
2D Example

Partition Section



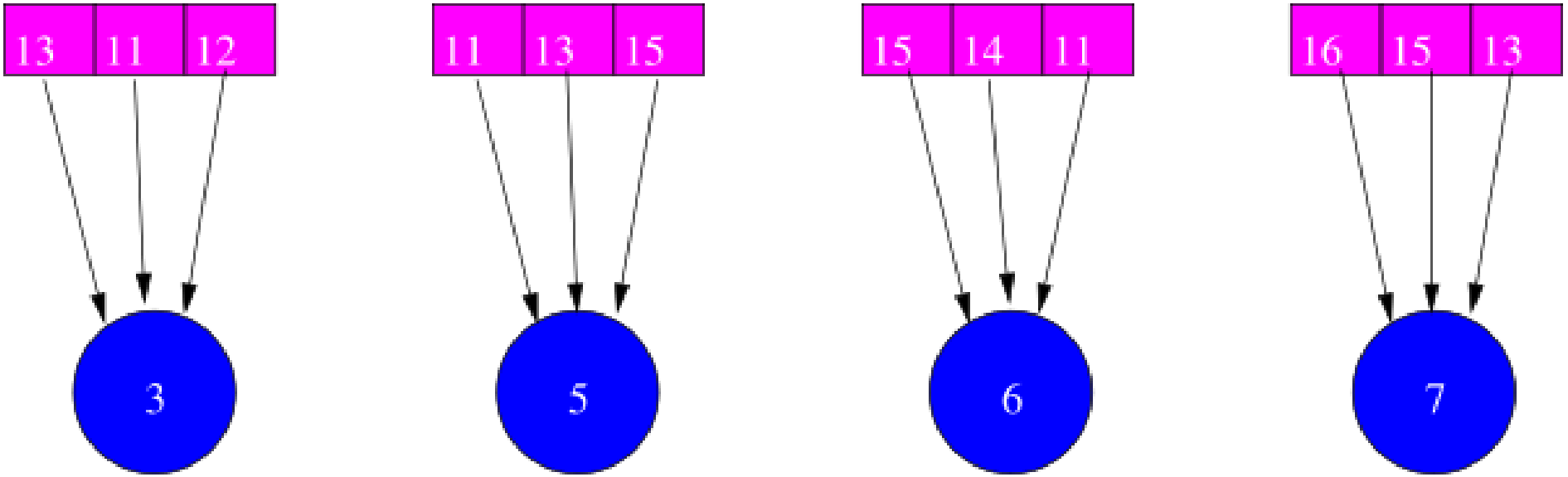
2D Example

Updated Sieve Overlap



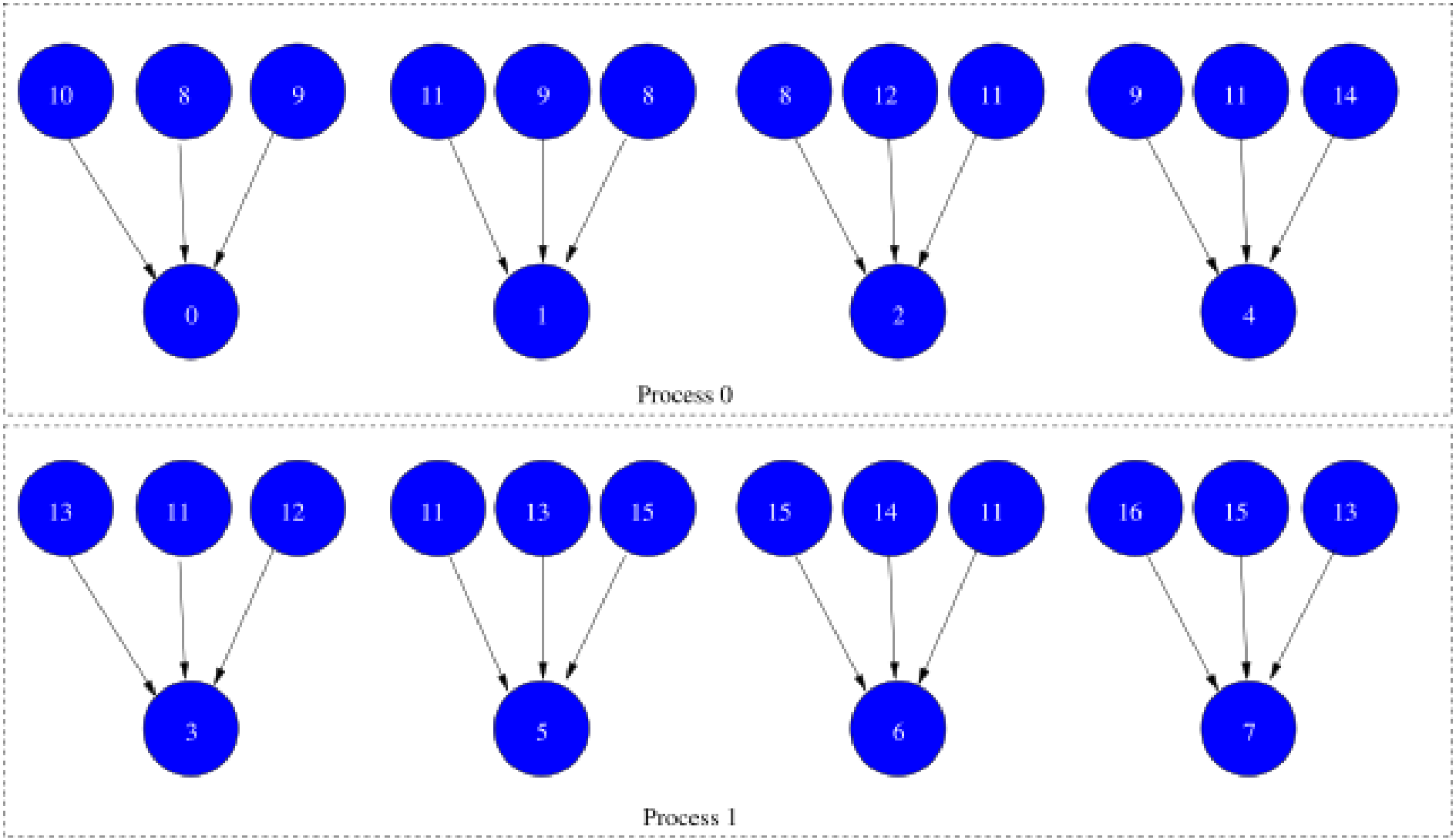
2D Example

Cone Section



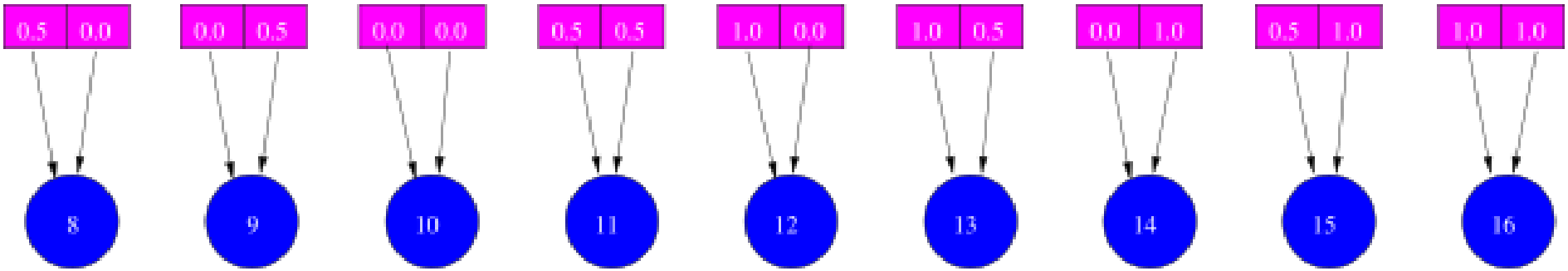
2D Example

Distributed Sieve



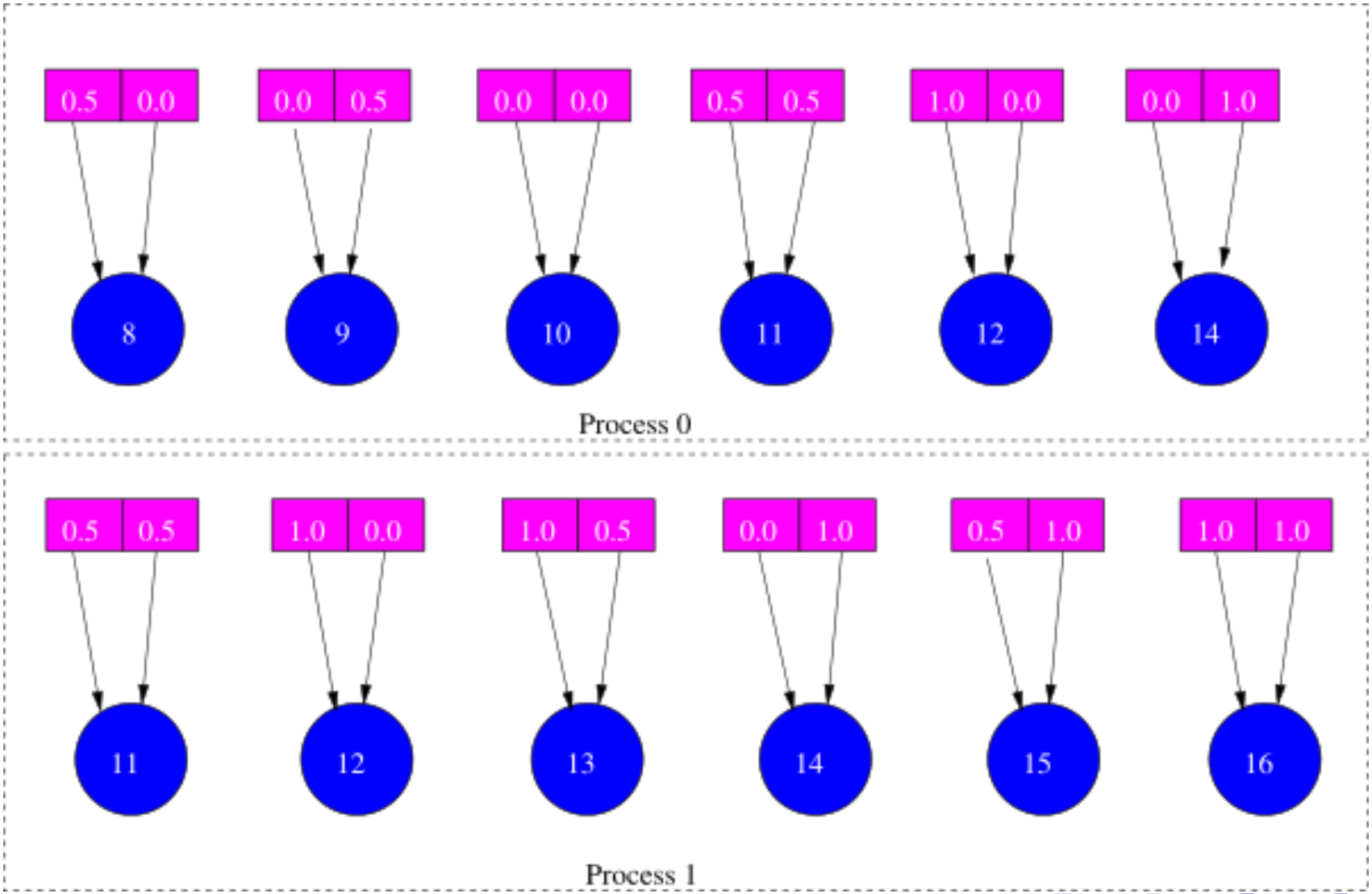
2D Example

Coordinate Section



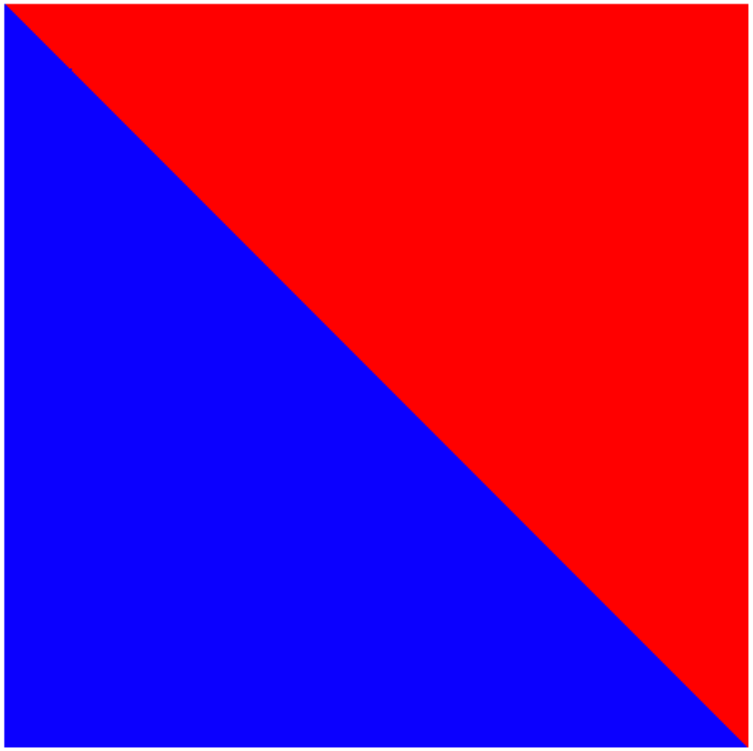
2D Example

Distributed Coordinate Section



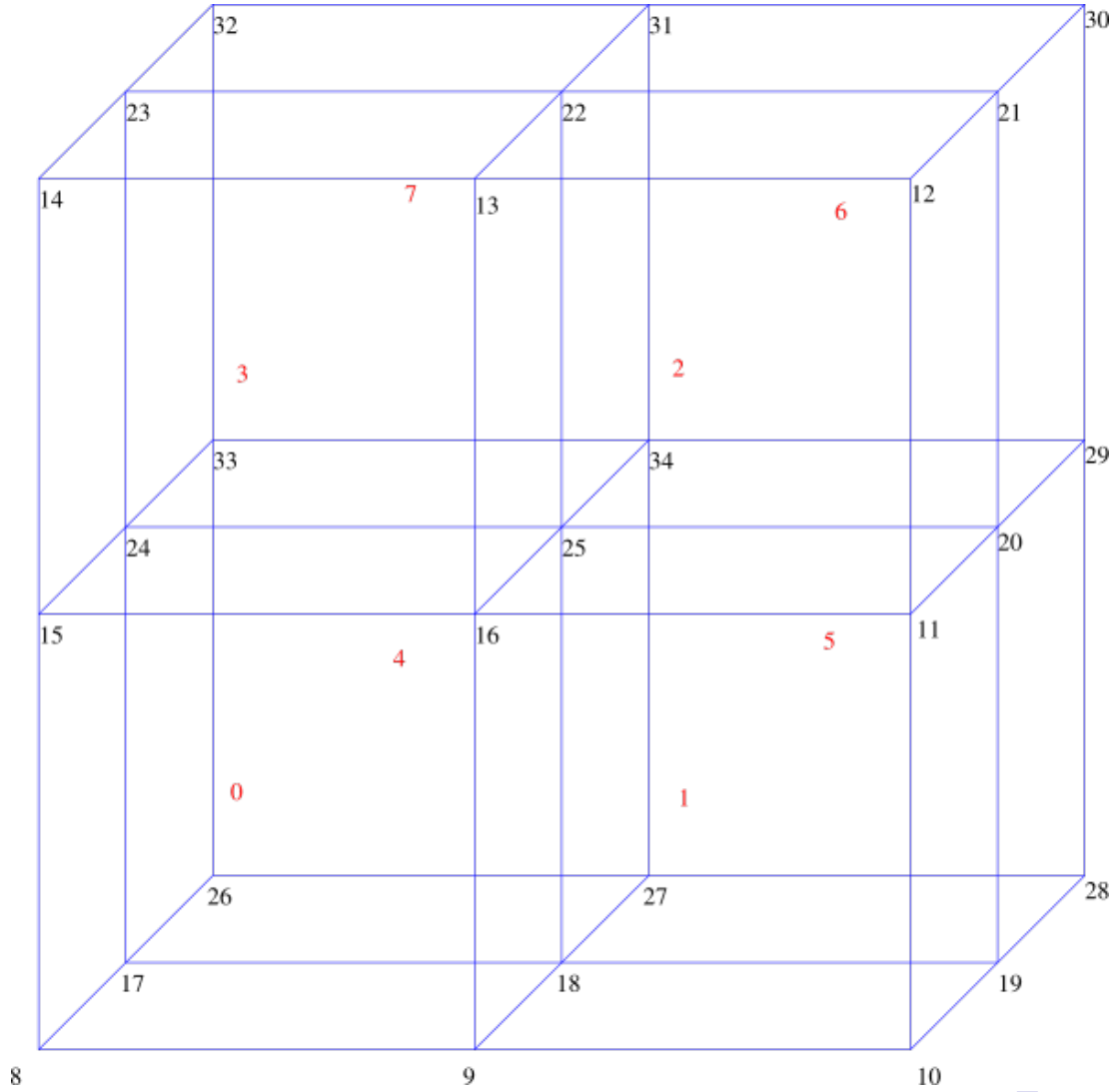
2D Example

Distributed Mesh



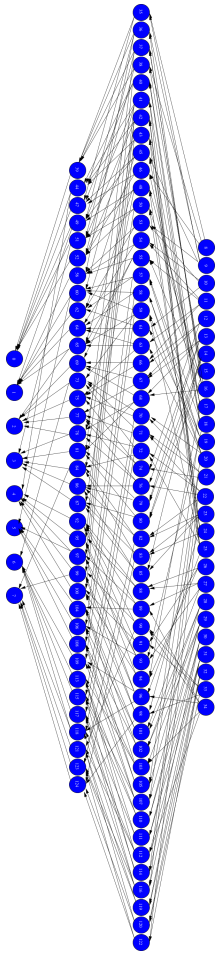
3D Example

A simple hexahedral mesh



3D Example

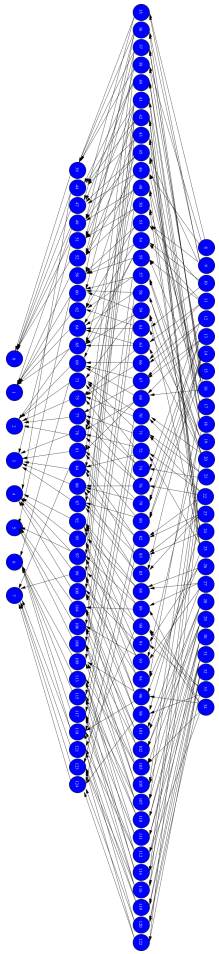
Sieve for the mesh



Its complicated!

3D Example

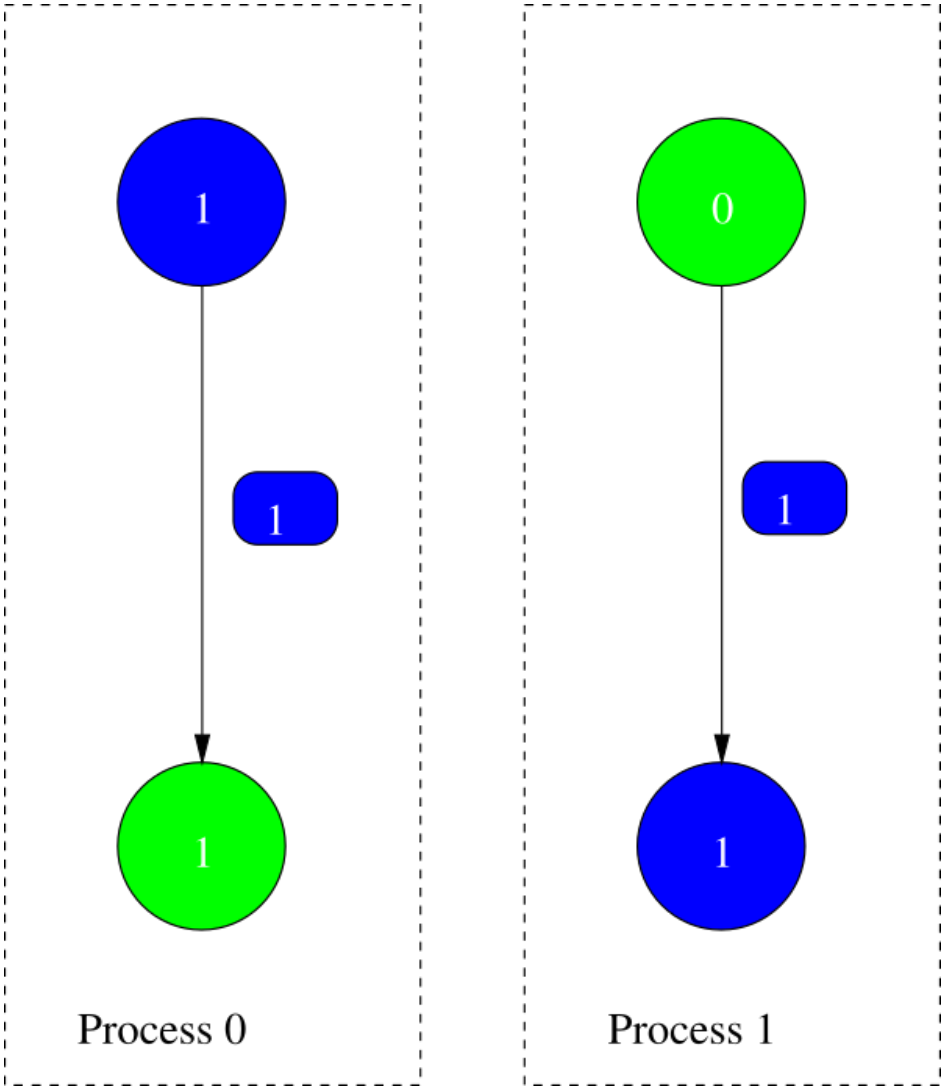
Sieve for the mesh



Its complicated!

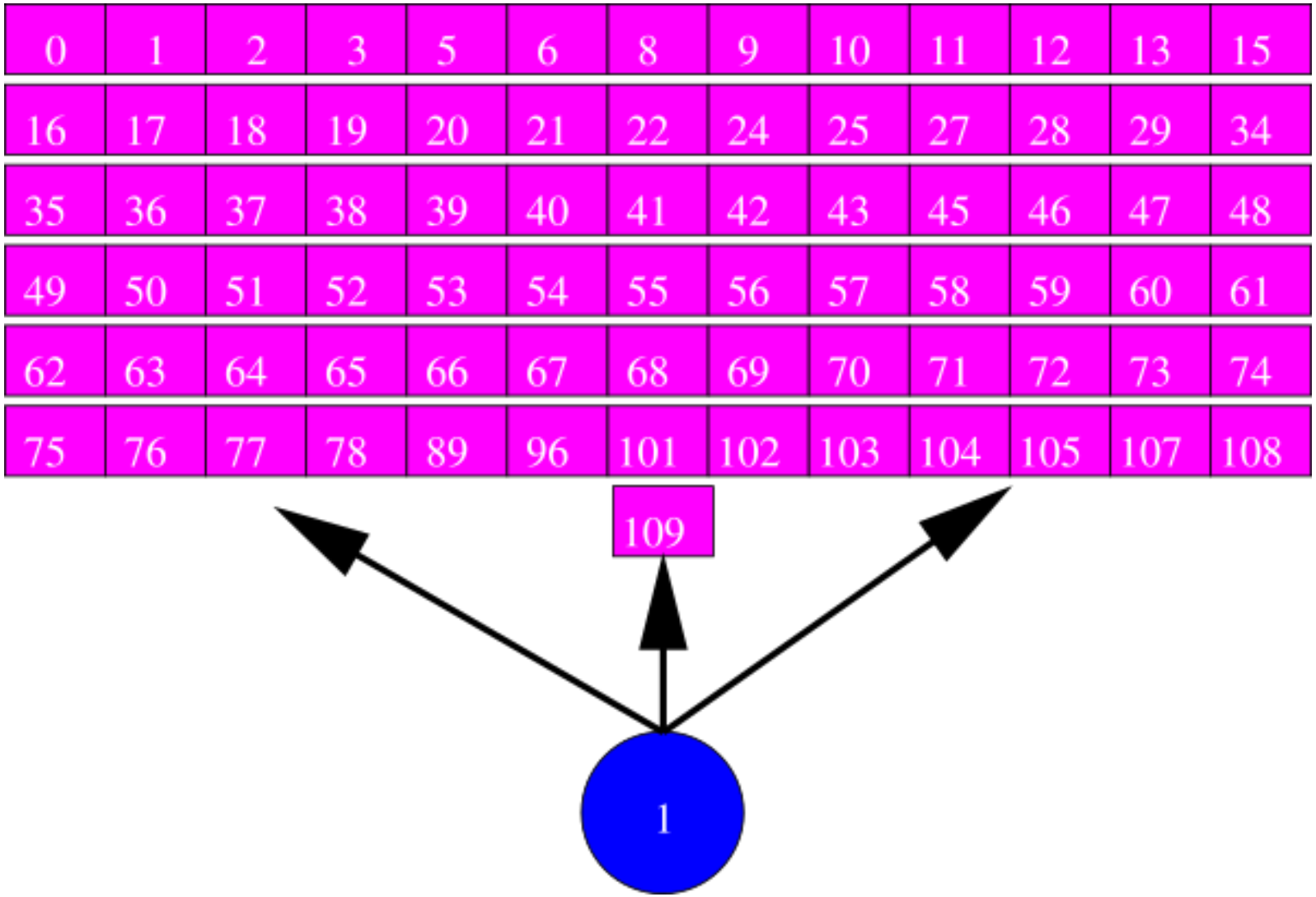
3D Example

Partition Overlap



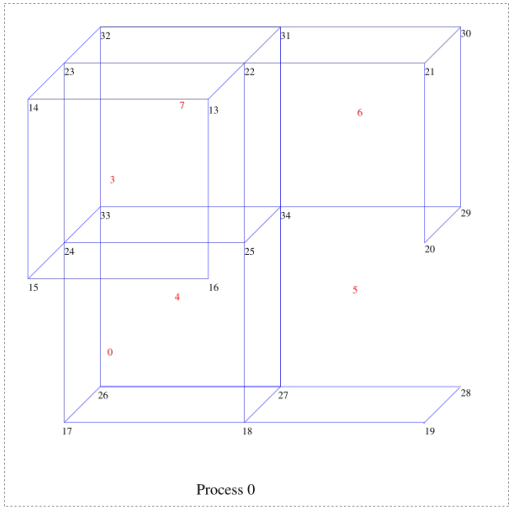
3D Example

Partition Section



3D Example

Distributed Mesh



Notice cells are ghosted

Outline

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

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)

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

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

```
for(c = cells->begin(); c != cells->end(); ++c) {
  SectionRestrictClosure(coordinates, dm, c, &coords);
  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*/
}
/* Aggregate updates */
```

Integration

```
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

```
for(c = cells->begin(); c != cells->end(); ++c) {
  /* Compute cell geometry */
  SectionRestrictClosure(U, dm, c, &inputVec);
  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

```
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

```
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

```
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

```
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

```
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

```
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 */
      /* Transform J */
      for(d = 0; d < dim; ++d)
        for(e = 0; e < dim; ++e)
          tDerReal[d] += invJ[e,d]*basisDer[q,f,e];
      for(g = 0; g < numBasisFuncs; ++g) {
        for(d = 0; d < dim; ++d)
          for(e = 0; e < dim; ++e)
            bDerReal[d] += invJ[e,d]*basisDer[q,g,e];
          /* Update element matrix */
        /* Update element vector */
      }
      /* Nonlinear term */
      elemVec[f] *= weight[q]*detJ;
    }
  }
  /* Update output vector*/
```

Integration

```
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 */
      /* Transform J */
      for(g = 0; g < numBasisFuncs; ++g) {
        for(d = 0; d < dim; ++d)
          elemMat[f,g] += tDerReal[d]*bDerReal[d];
        elemVec[f] += elemMat[f,g]*inputVec[g];
      }
      /* Nonlinear term */
      elemVec[f] *= weight[q]*detJ;
    }
  }
  /* Update output vector*/
}
/* Aggregate updates */
```

Integration

```
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

```
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

```
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

```
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;
    }
  }
  SectionRealUpdate(locF, c, elemVec, ADD_VALUES);
}
/* Aggregate updates */
```

Integration

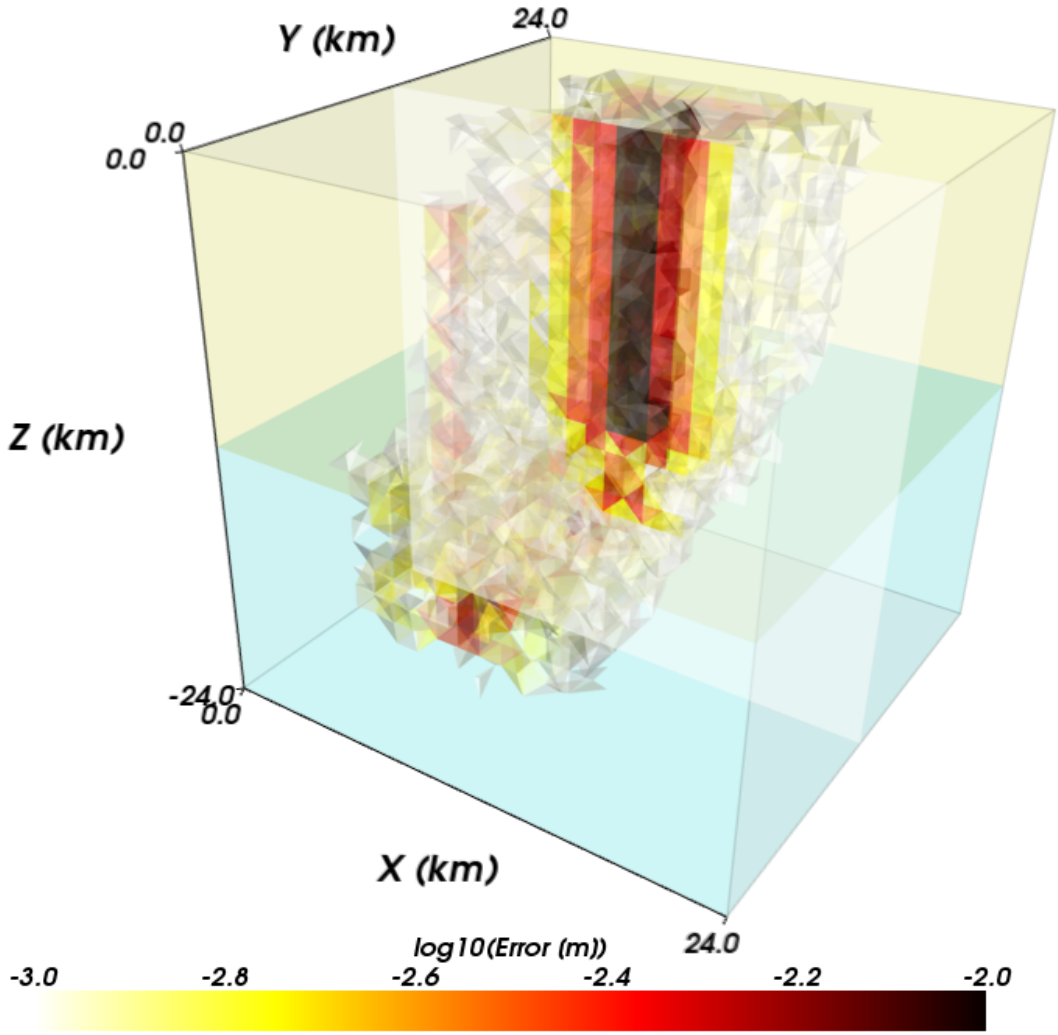
```
for(c = cells->begin(); c != cells->end(); ++c) {
  /* Compute cell geometry */
  /* Retrieve values from input vector */
  for(q = 0; q < numQuadPoints; ++q) {
    /* Transform coordinates */
    for(f = 0; f < numBasisFuncs; ++f) {
      /* Constant term */
      /* Linear term */
      /* Nonlinear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

Integration

```
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*/
}
DMLocalToGlobalBegin(dm, locF, INSERT_VALUES, F);
DMLocalToGlobalEnd(dm, locF, INSERT_VALUES, F);
```

PyLith

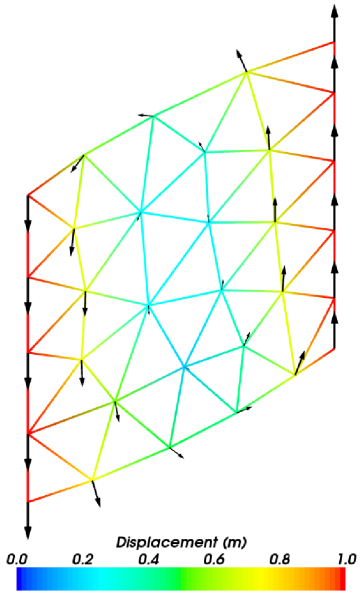
- Multiple problems
  - Dynamic rupture
  - Quasi-static relaxation
- Multiple models
  - Nonlinear visco-plastic
  - Finite deformation
  - Fault constitutive models
- Multiple meshes
  - 1D, 2D, 3D
  - Hex and tet meshes
- Parallel
  - PETSc solvers
  - DMPlex mesh management

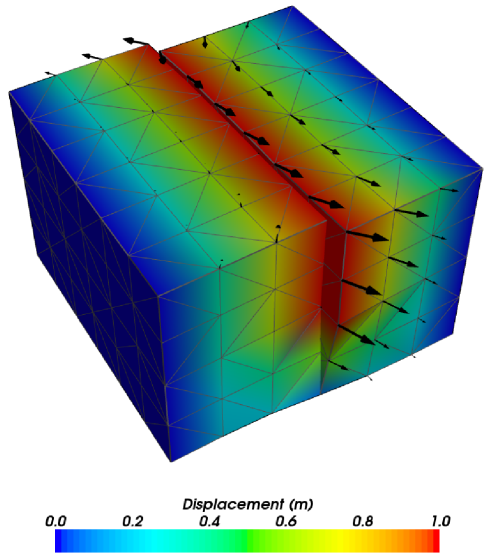


[a] Aagaard, Knepley, Williams

Multiple Mesh Types

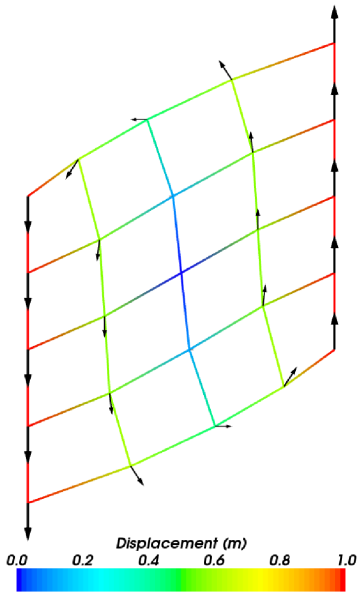
Triangular

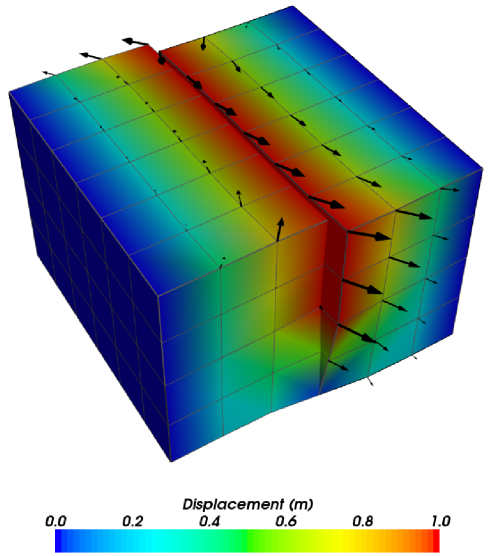




Tetrahedral

Rectangular

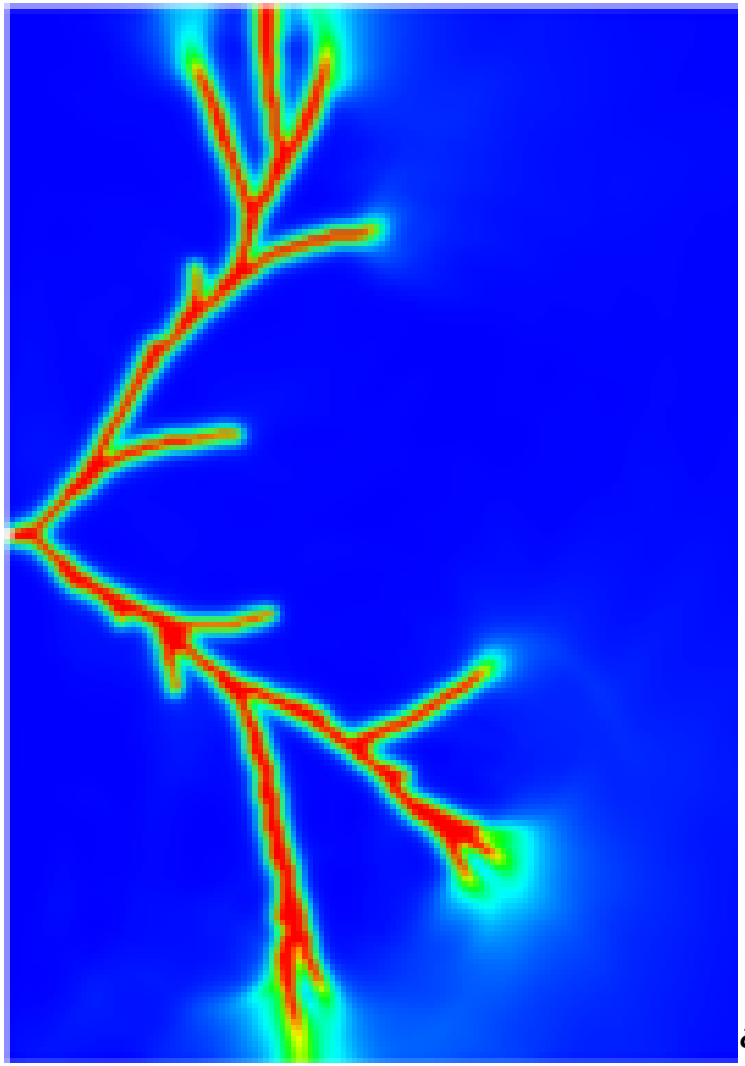




Hexahedral

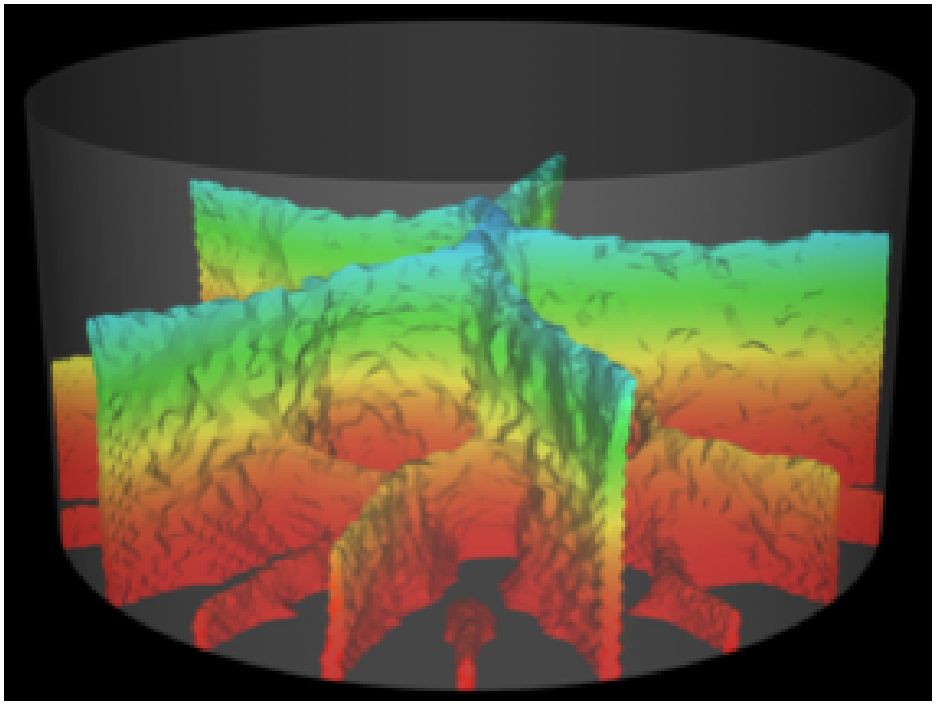
Fracture Mechanics

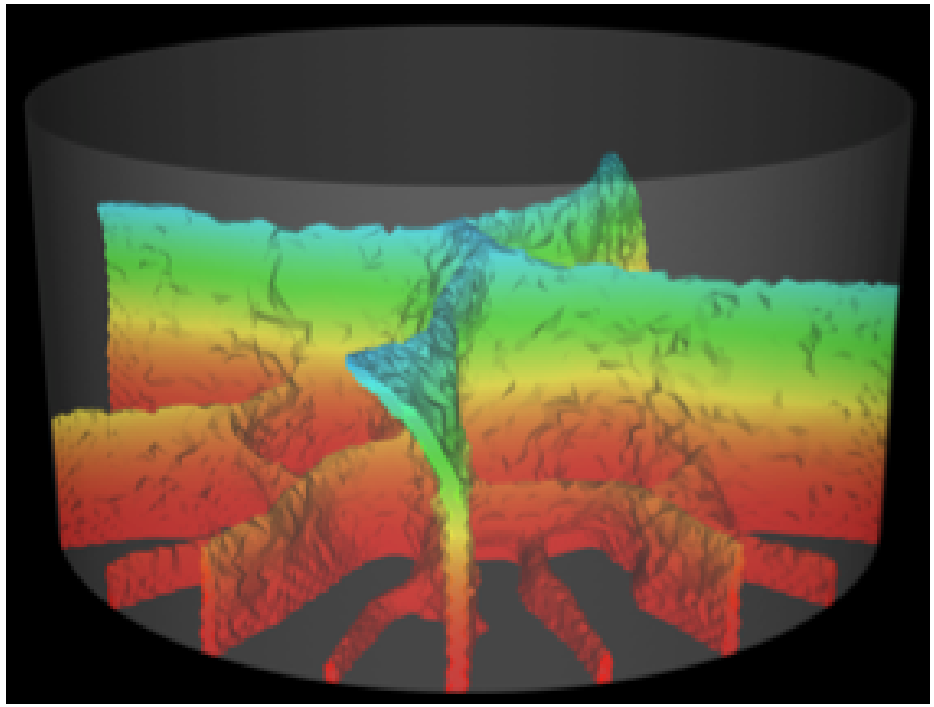
- Full variational formulation
  - Phase field
  - Linear or Quadratic penalty
- Uses TAO optimization
  - Necessary for linear penalty
  - Backtacking
- No prescribed cracks (movie)
  - Arbitrary crack geometry
  - Arbitrary intersections
- Multiple materials
  - Composite toughness



[a] Bourdin

Fracture Mechanics





[1]

[1] Bourdin

Outline

Main Point

Using estimates and proofs,

a simple software architecture,

achieves good scaling

and adaptive load balance.

Main Point

Using estimates and proofs,

a simple software architecture,

achieves good scaling

and adaptive load balance.

Main Point

Using estimates and proofs,
a simple software architecture,
achieves good scaling
and adaptive load balance.

Outline

FMM Applications

FMM can accelerate both integral and boundary element methods for:

- Laplace
- Stokes
- Elasticity

FMM Applications

FMM can accelerate both integral and boundary element methods for:

- Laplace
- Stokes
- Elasticity

Advantages

- Mesh-free
- $\mathcal{O}(N)$ time
- Distributed and multicore (GPU) parallelism
- Small memory bandwidth requirement

Fast Multipole Method

FMM accelerates the calculation of the function:

$$\Phi(x_i) = \sum_j K(x_i, x_j) q(x_j) \tag{8}$$

- Accelerates $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ time
- The kernel $K(x_i, x_j)$ must decay quickly from (x_i, x_i)
  - Can be singular on the diagonal (Calderón-Zygmund operator)
- Discovered by Leslie Greengard and Vladimir Rohklin in 1987
- Very similar to recent wavelet techniques

Fast Multipole Method

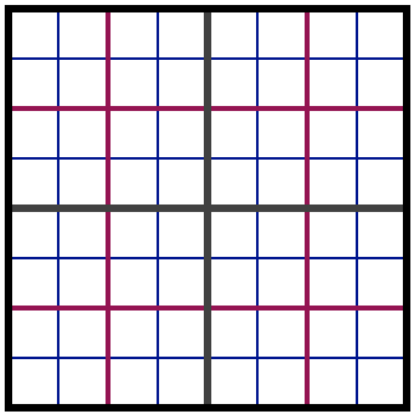
FMM accelerates the calculation of the function:

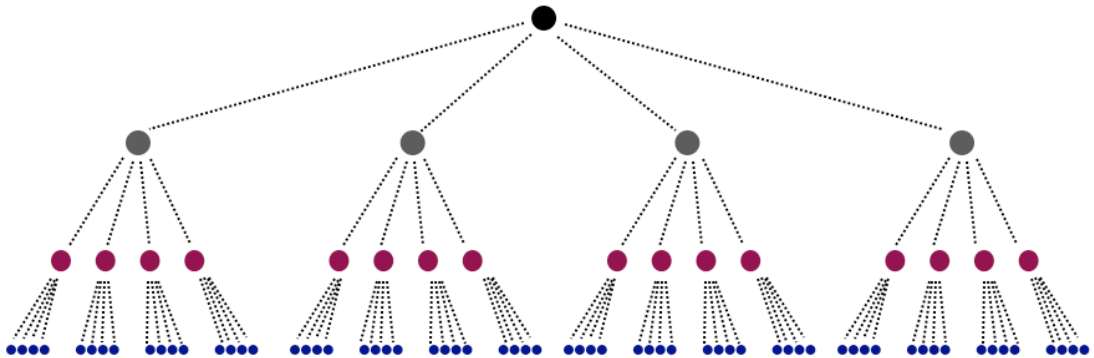
$$\Phi(x_i) = \sum_j \frac{q_j}{|x_i - x_j|} \tag{8}$$

- Accelerates $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ time
- The kernel $K(x_i, x_j)$ must decay quickly from (x_i, x_i)
  - Can be singular on the diagonal (Calderón-Zygmund operator)
- Discovered by Leslie Greengard and Vladimir Rohklin in 1987
- Very similar to recent wavelet techniques

Spatial Decomposition

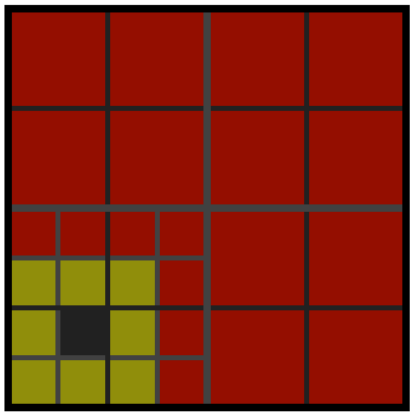
Pairs of boxes are divided into *near* and *far*:

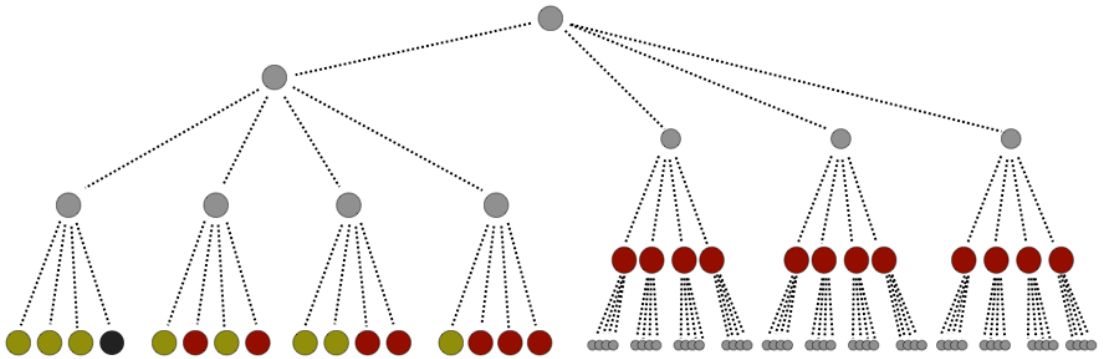




Spatial Decomposition

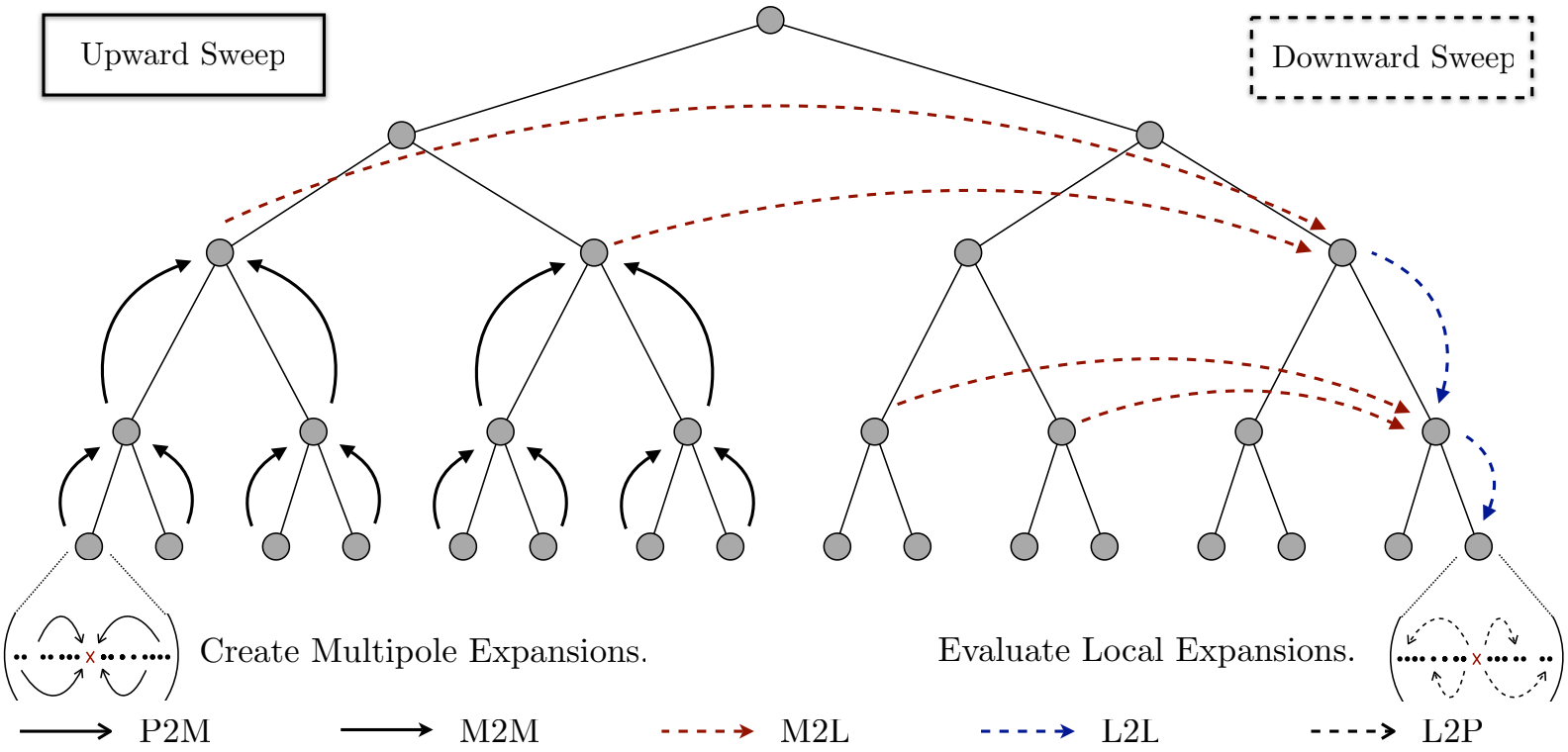
Pairs of boxes are divided into *near* and *far*:





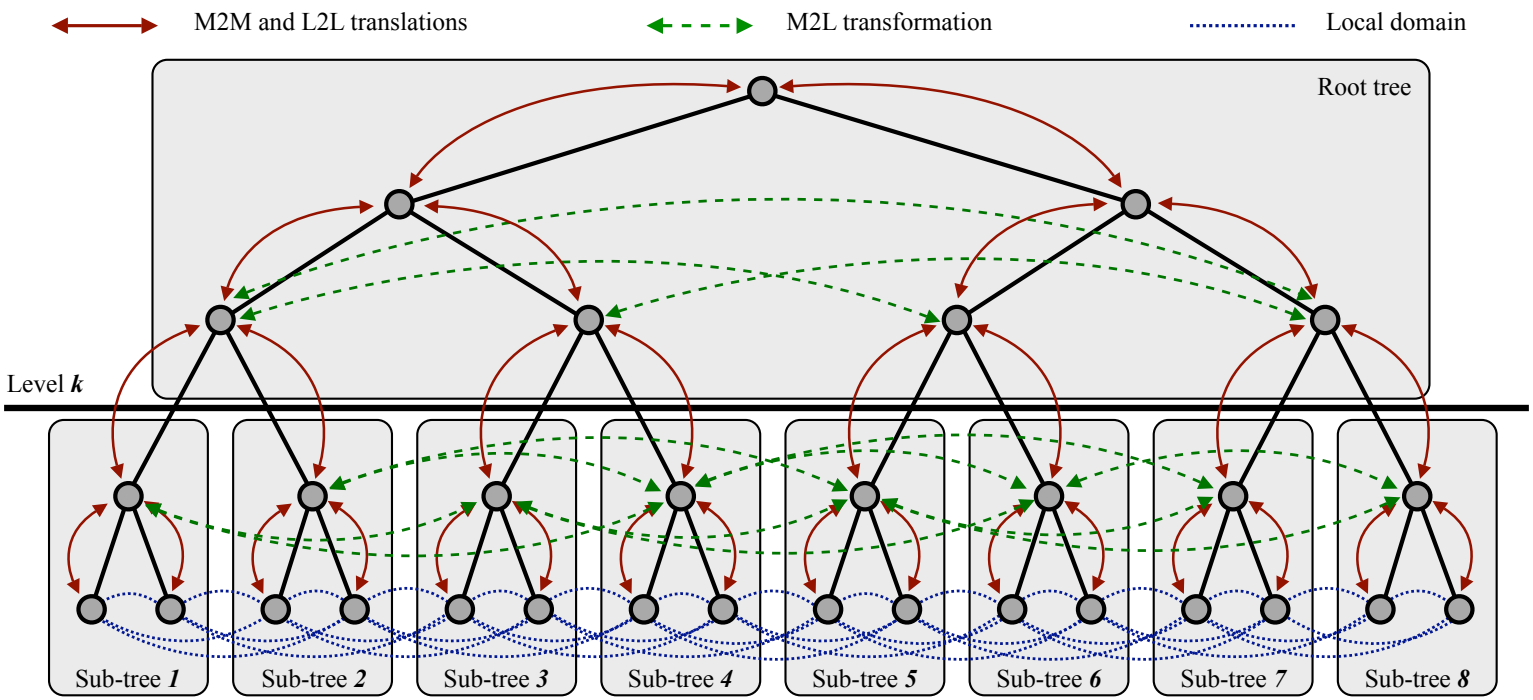
Neighbors are treated as *very near*.

FMM Control Flow



Kernel operations will map to GPU tasks.

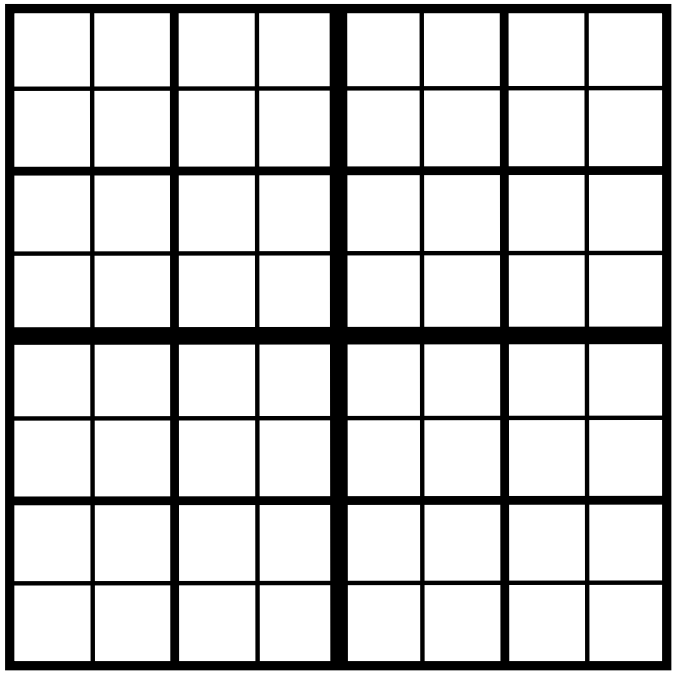
FMM Control Flow
Parallel Operation



Kernel operations will map to GPU tasks.

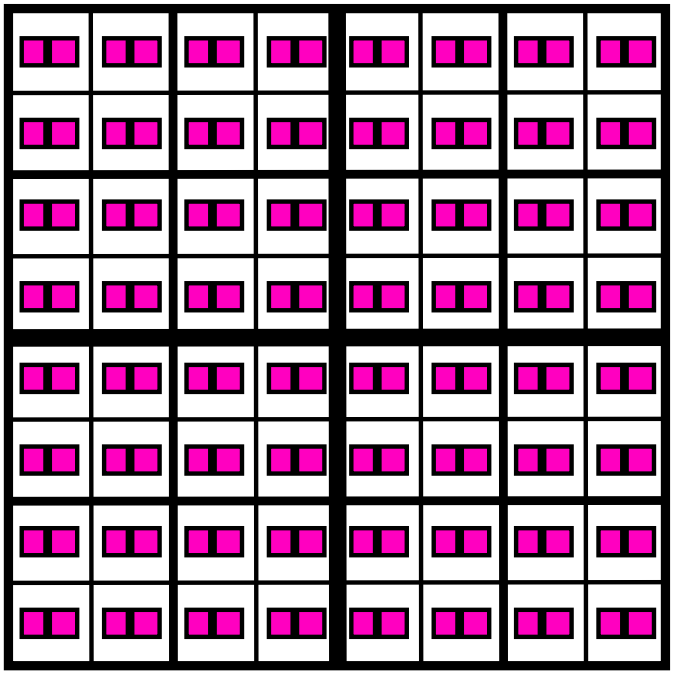
Outline

FMM in Sieve



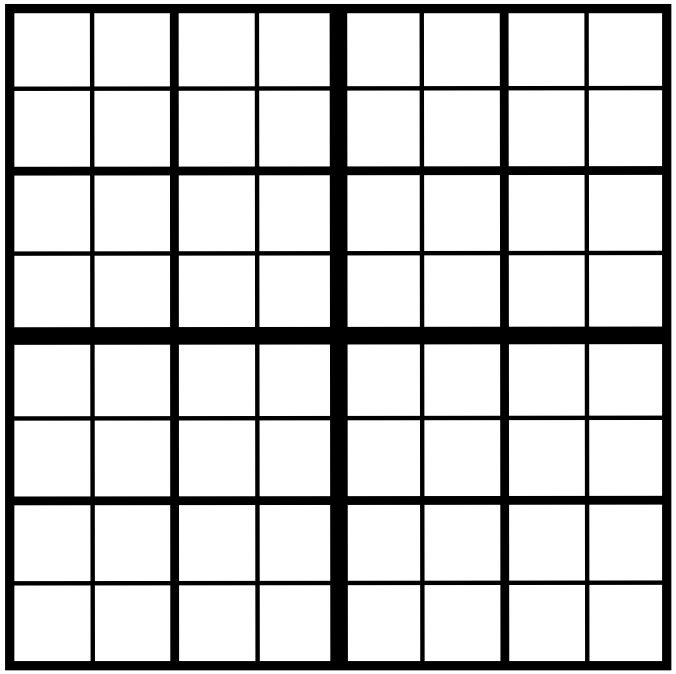
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



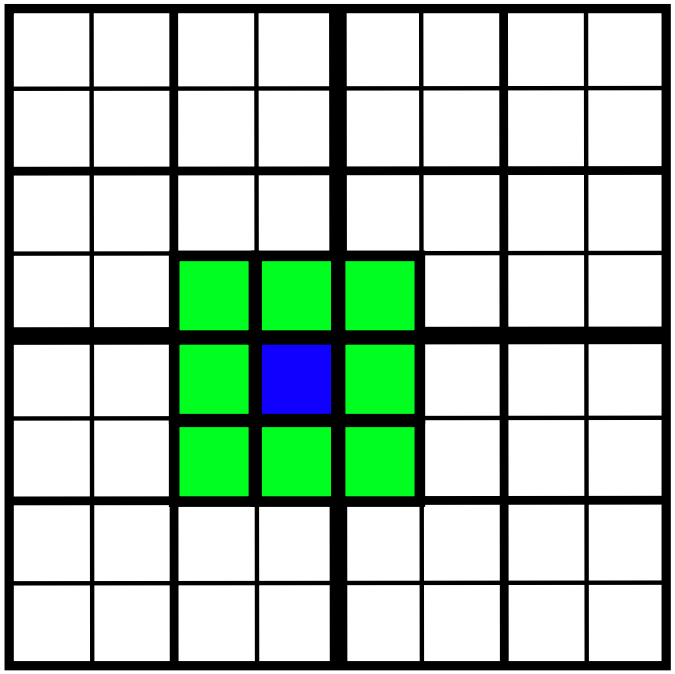
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



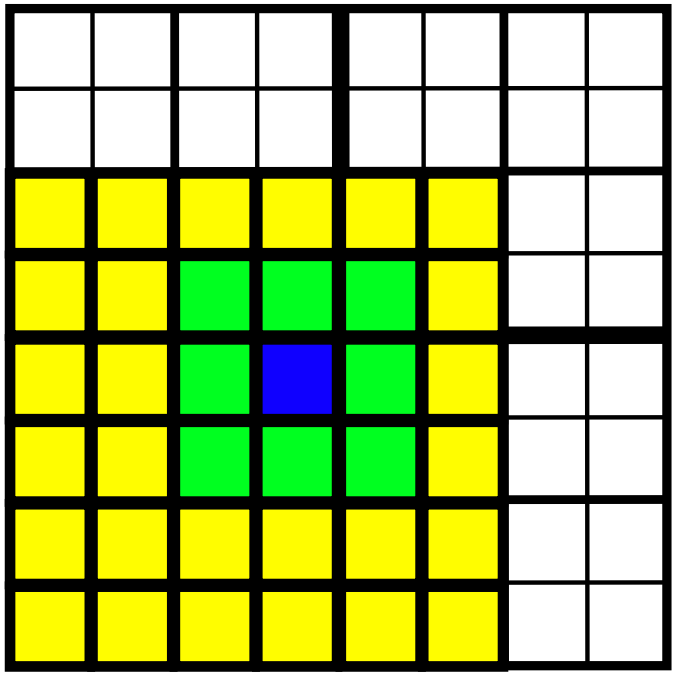
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



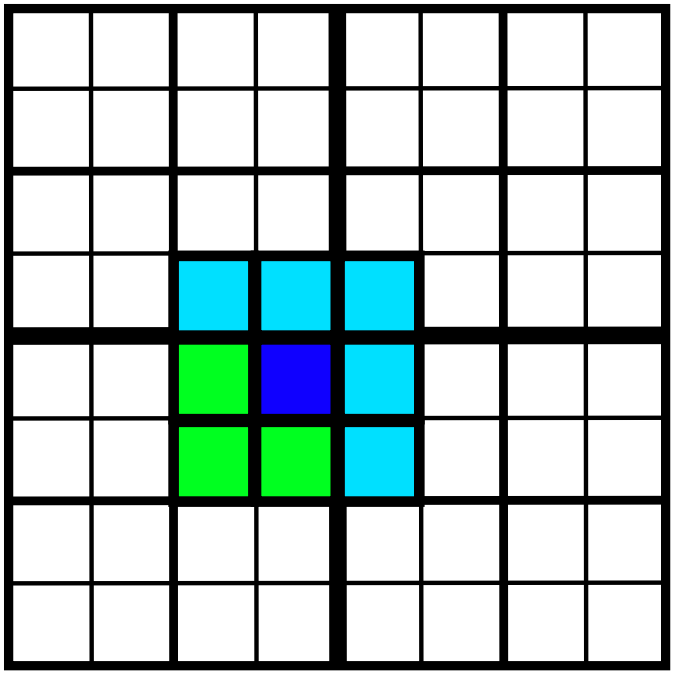
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



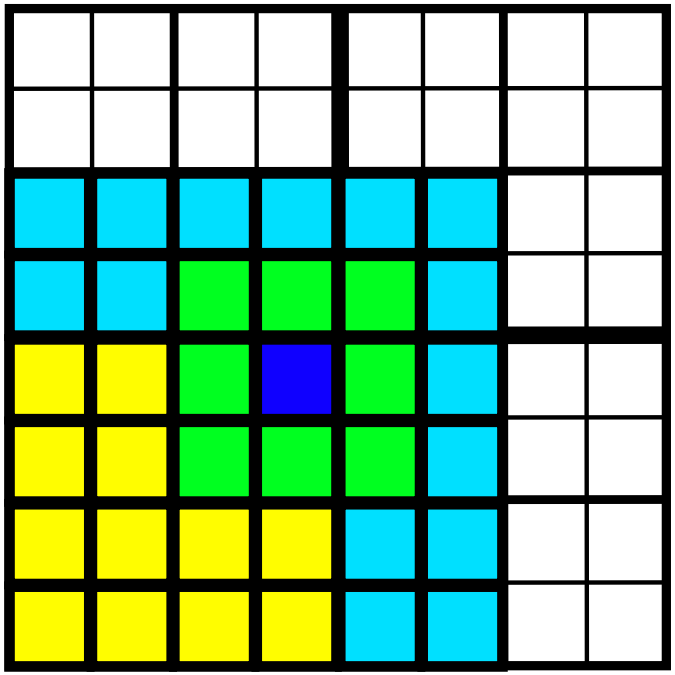
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



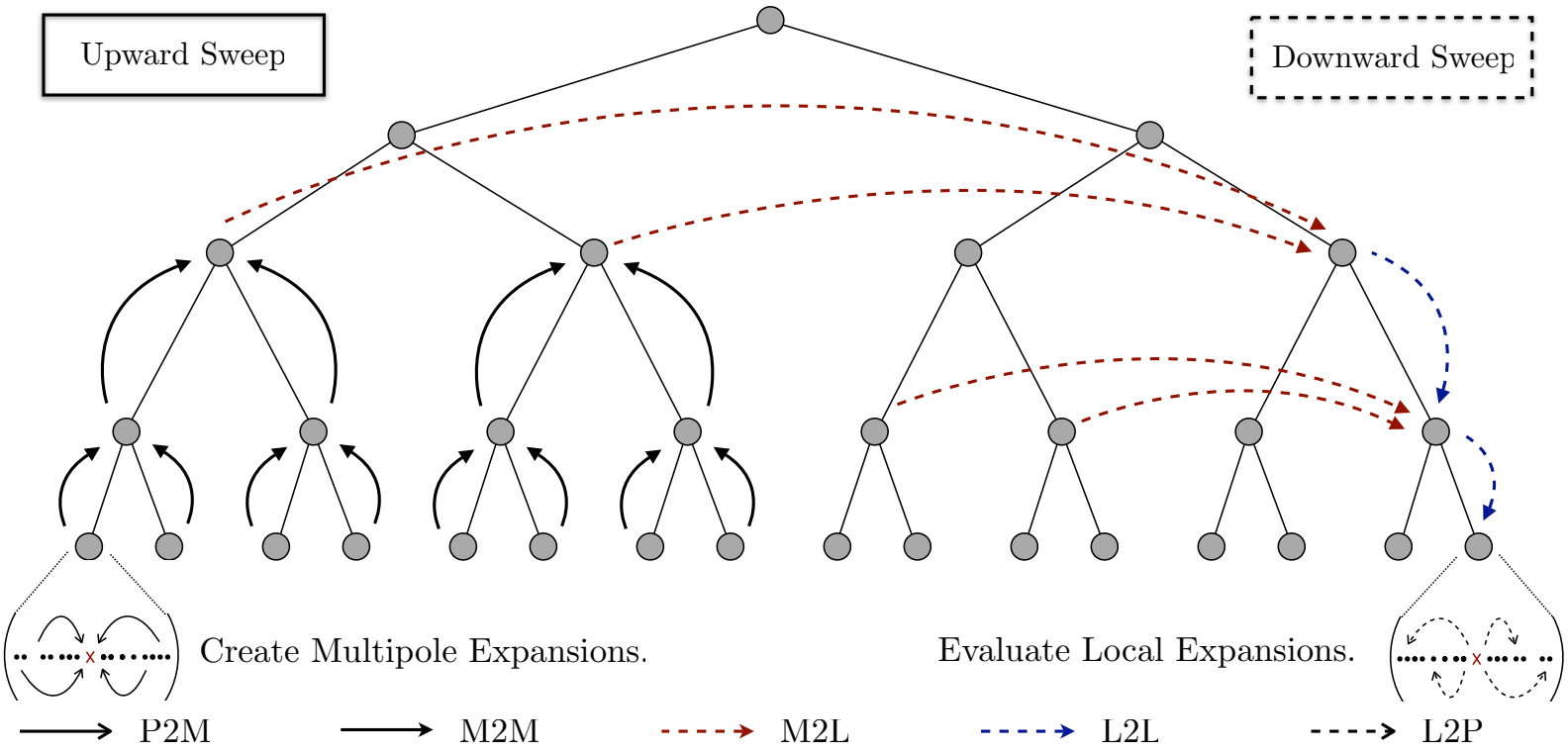
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



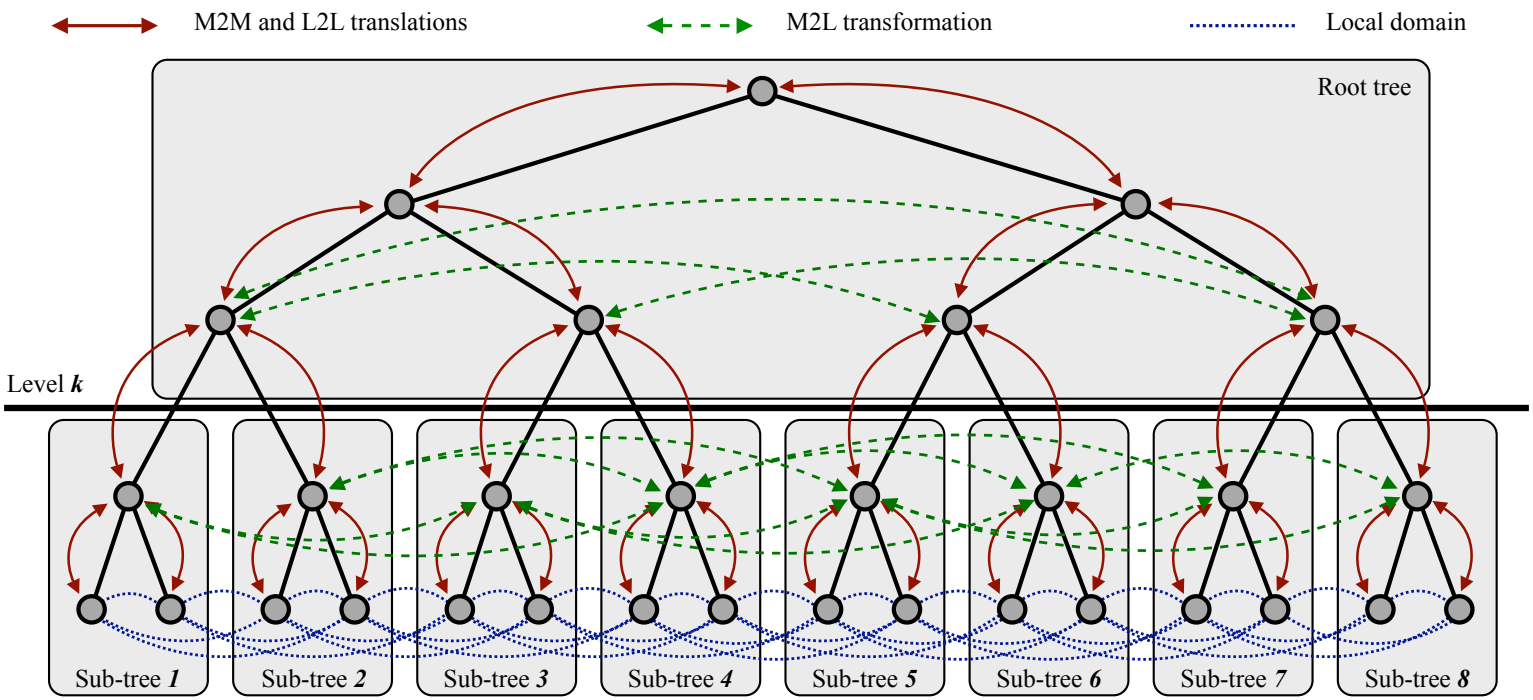
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM Control Flow



Kernel operations will map to GPU tasks.

FMM Control Flow
Parallel Operation



Kernel operations will map to GPU tasks.

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

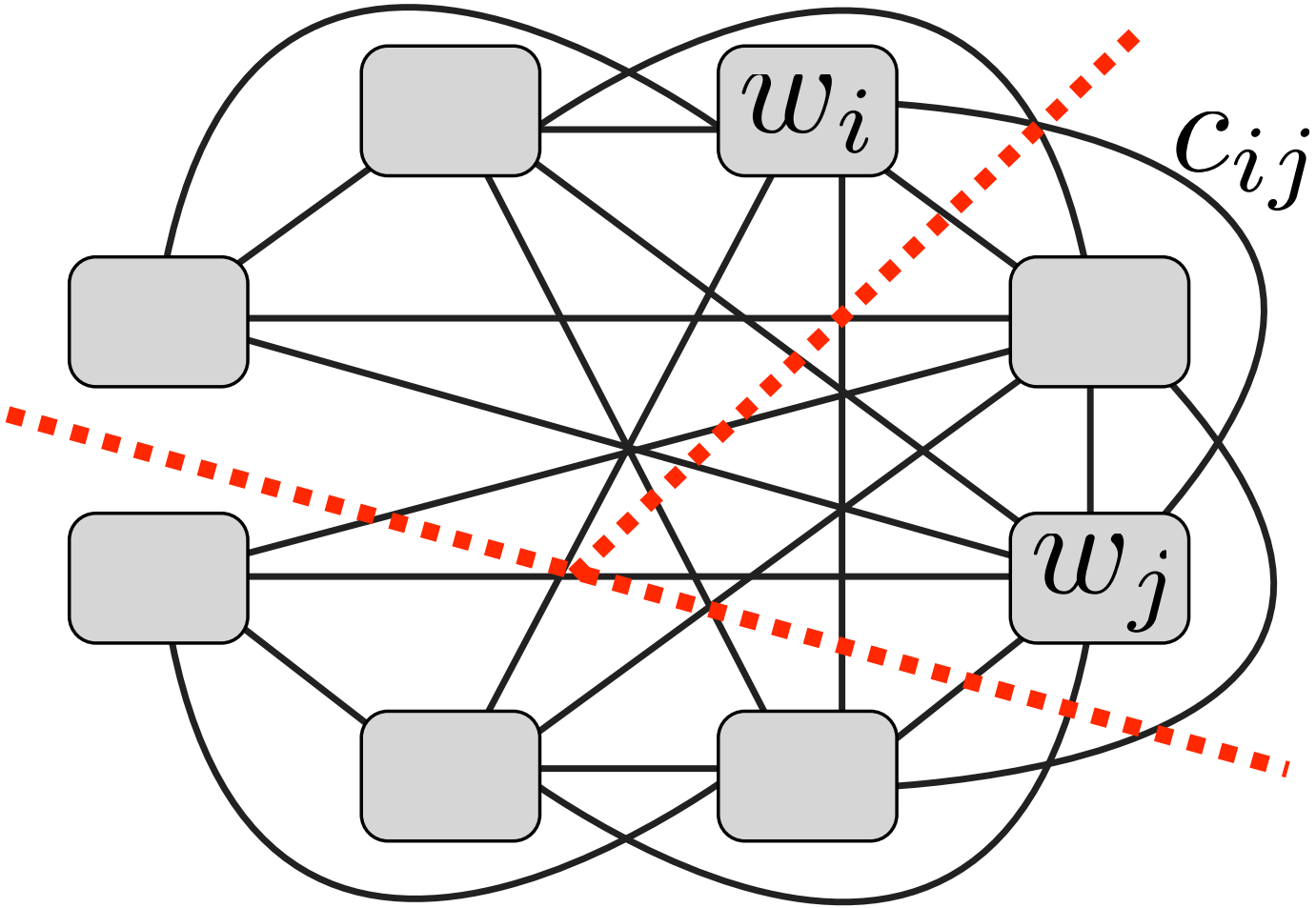
- Simplicity
- Complete serial code reuse
- Provably good performance and scalability

Parallel Tree Implementation
Advantages

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

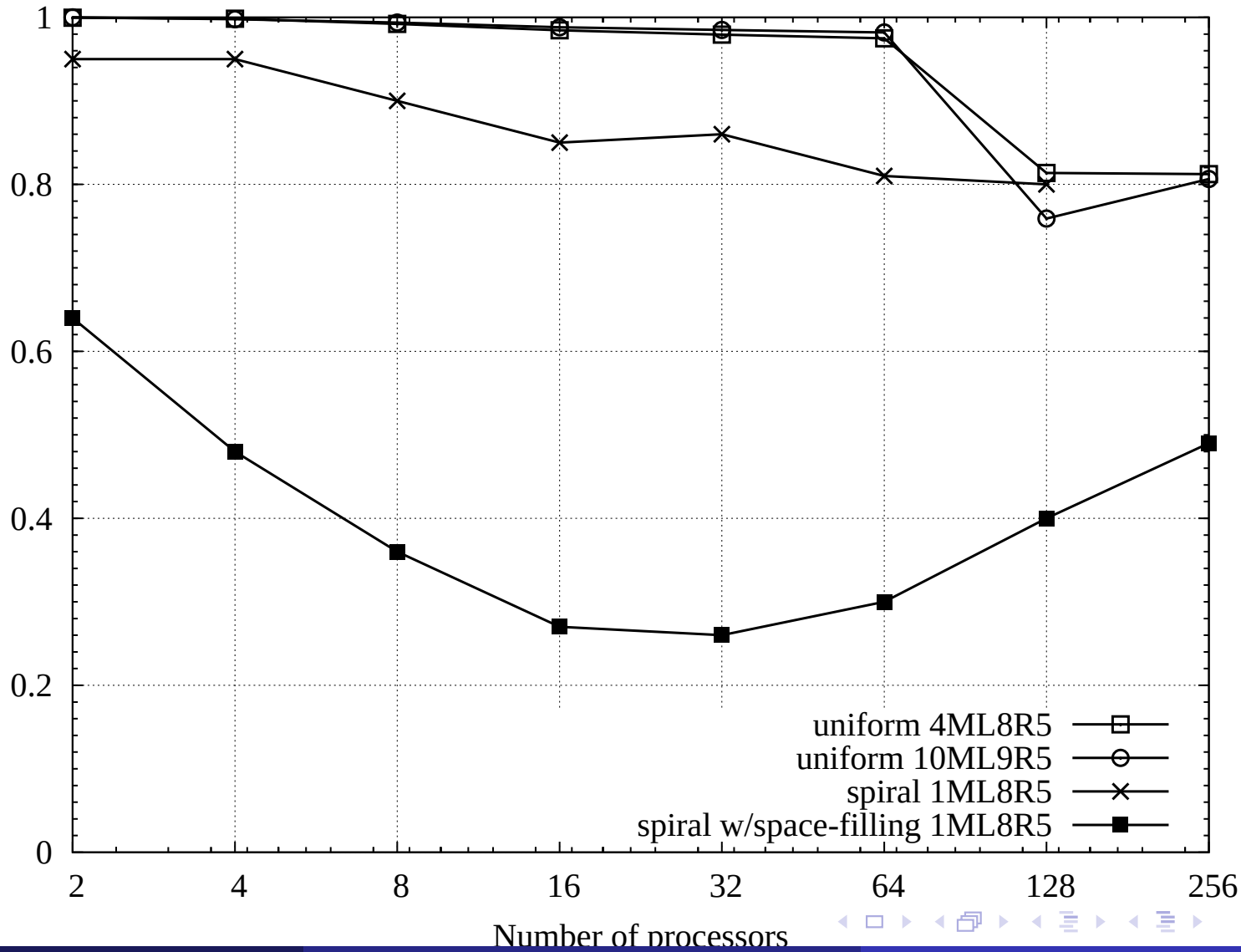
Distributing Local Trees

The interaction of locals trees is represented by a weighted graph.



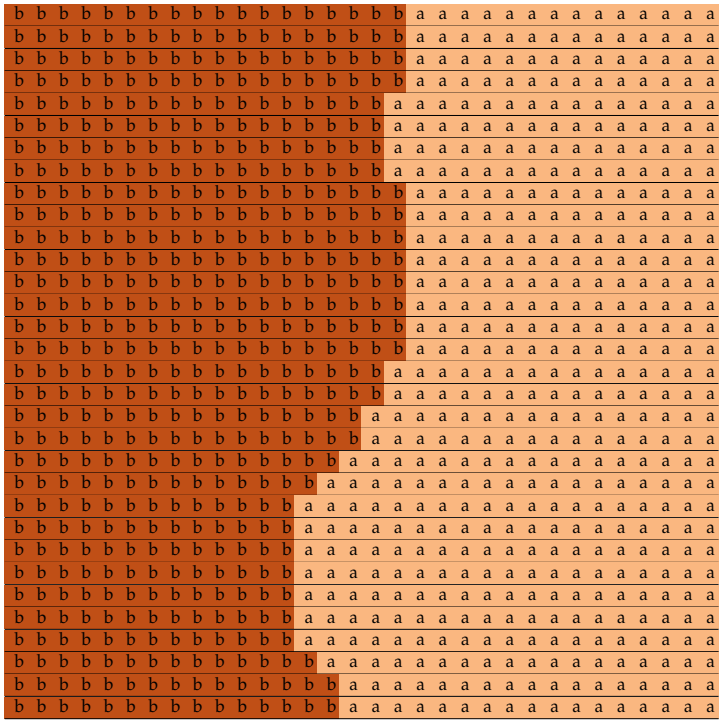
This graph is partitioned, and trees assigned to processes.

PetFMM Load Balance

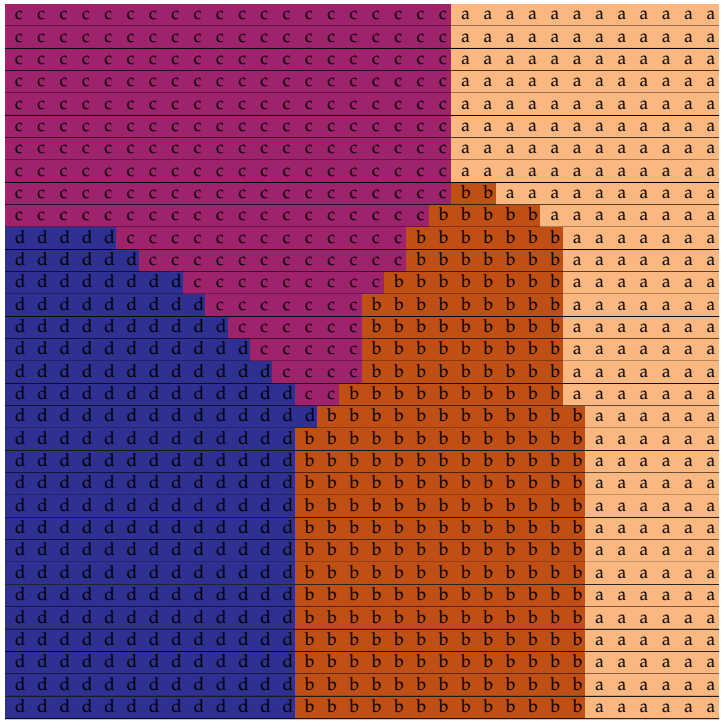


Local Tree Distribution

Here local trees are assigned to processes for a spiral distribution:



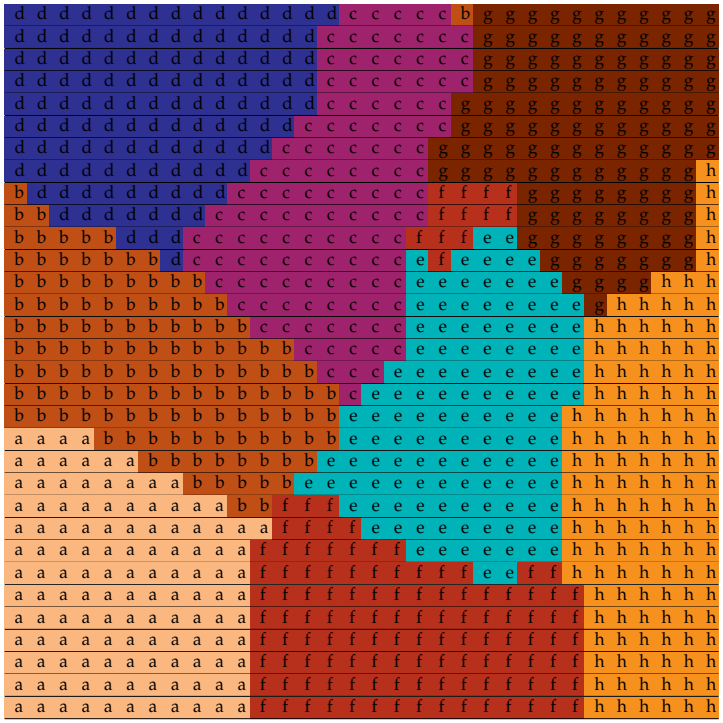
(a) 2 cores



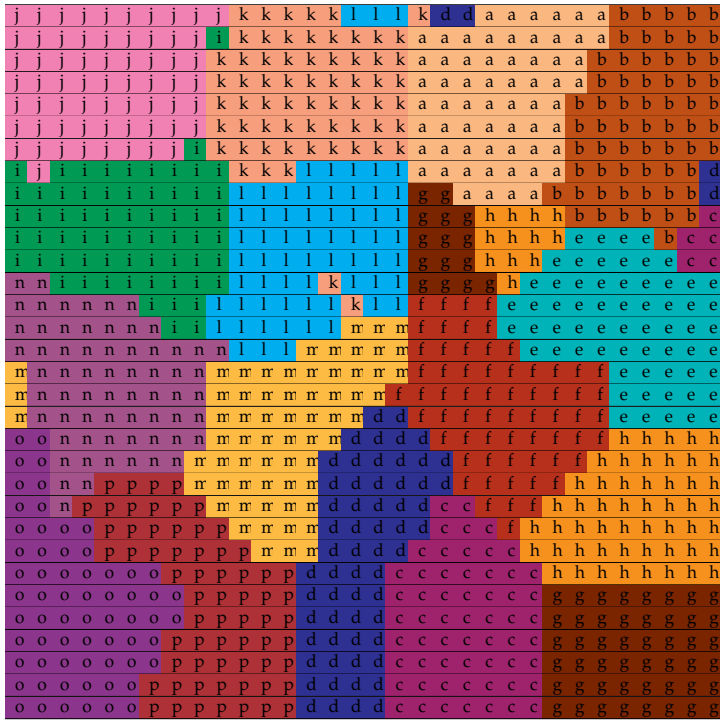
(b) 4 cores

Local Tree Distribution

Here local trees are assigned to processes for a spiral distribution:



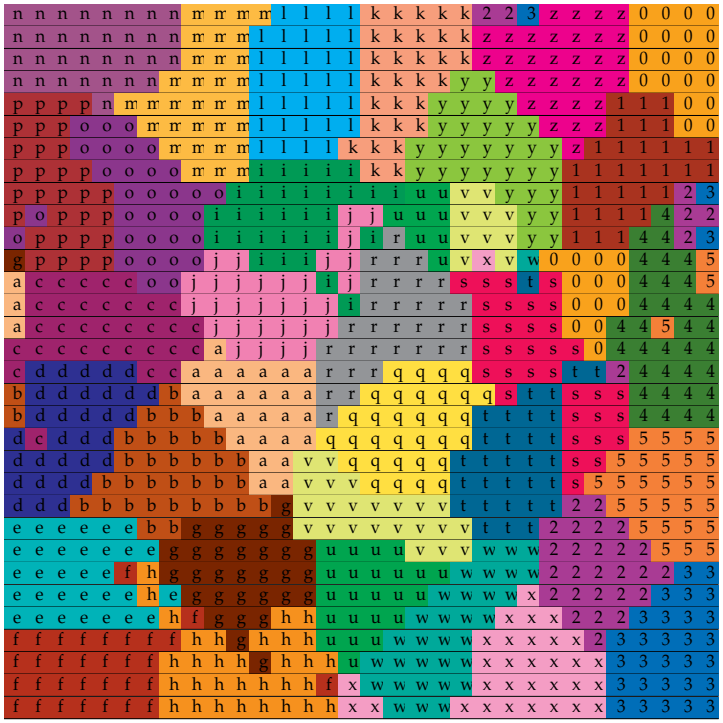
(c) 8 cores



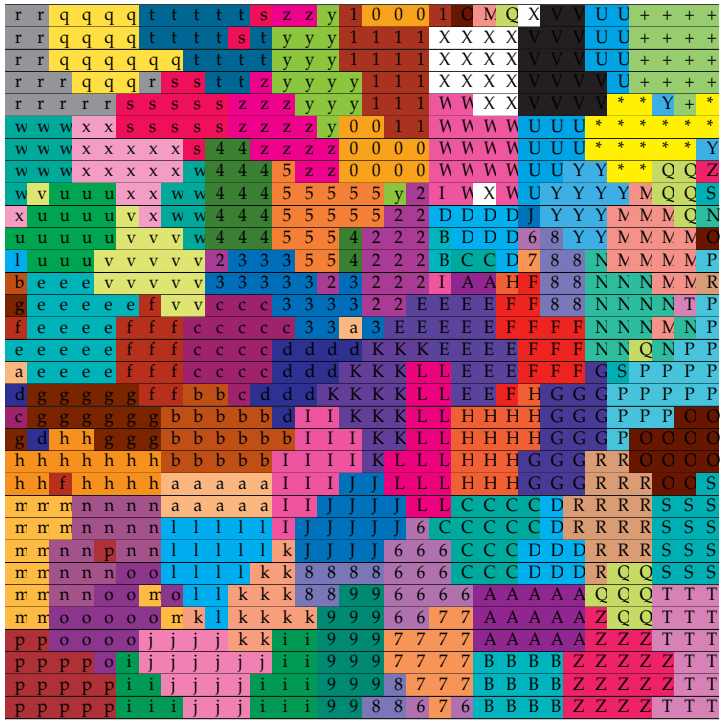
(d) 16 cores

Local Tree Distribution

Here local trees are assigned to processes for a spiral distribution:



(e) 32 cores



(f) 64 cores

Outline

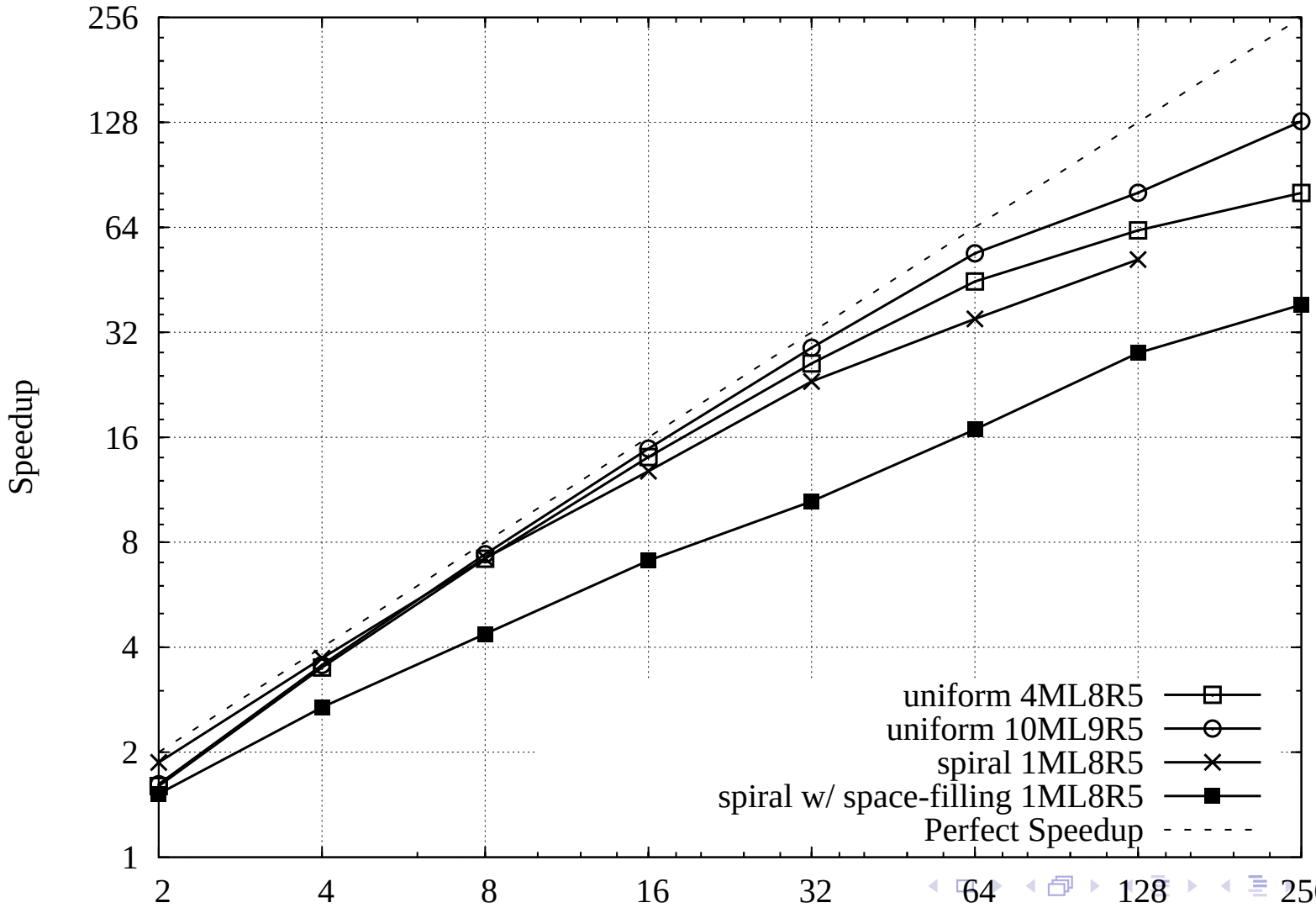
PetFMM

PetFMM is an freely available implementation of the
Fast Multipole Method
http://barbagroup.bu.edu/Barba_group/PetFMM.html

- Leverages PETSc
  - Same open source license
  - Uses Sieve for parallelism
- Extensible design in C++
  - Templated over the kernel
  - Templated over traversal for evaluation
- MPI implementation
  - Novel parallel strategy for anisotropic/sparse particle distributions
  - PetFMM–A dynamically load-balancing parallel fast multipole library
  - 86% efficient strong scaling on 64 procs
- Example application using the Vortex Method for fluids
- (coming soon) GPU implementation

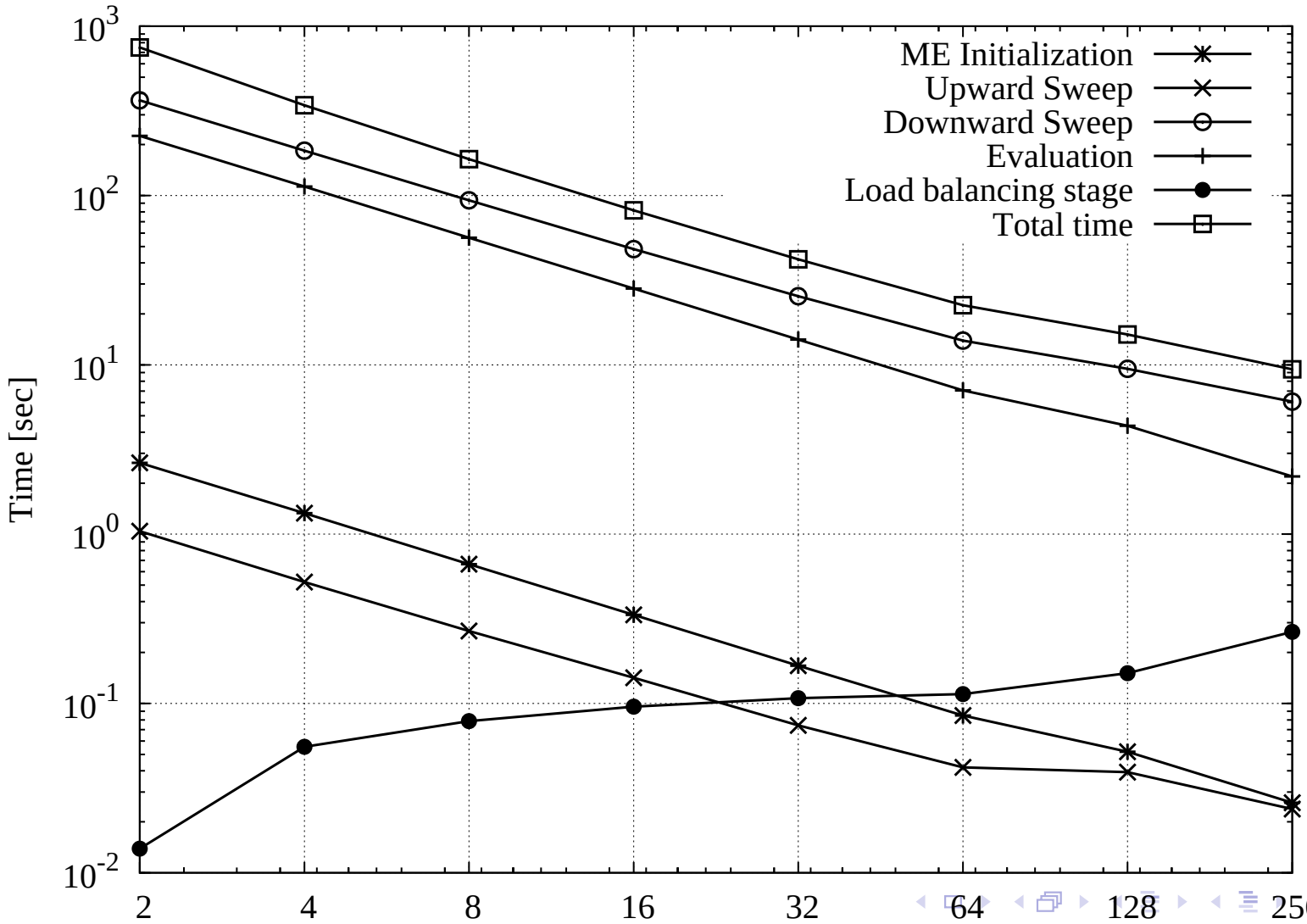
PetFMM CPU Performance

Strong Scaling



PetFMM CPU Performance

Strong Scaling



Conclusions

Better mathematical abstractions bring concrete benefits

- Vast reduction in complexity
  - Dimension and mesh independent code
  - Complete serial code reuse
- Opportunites for optimization
  - Higher level operations missed by traditional compilers
  - Single communication routine to optimize
- Expansion of capabilities
  - Arbitrary elements
  - Unstructured multigrid
  - Multilevel algorithms

Outline

- FEM
- UMG
- PyLith

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

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
```

Outline

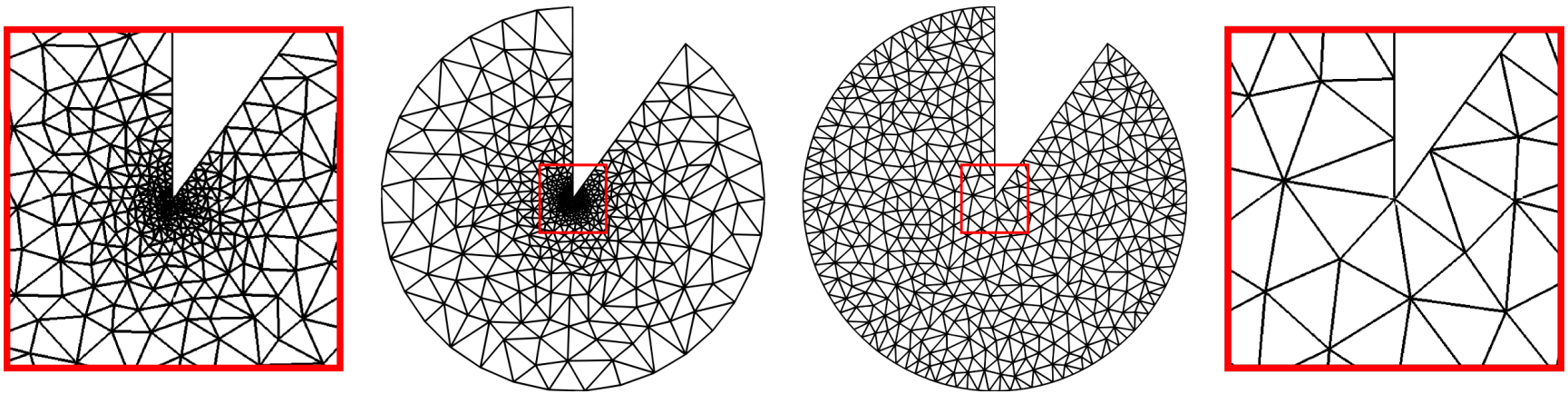
- FEM
- UMG
- PyLith

A Priori refinement

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

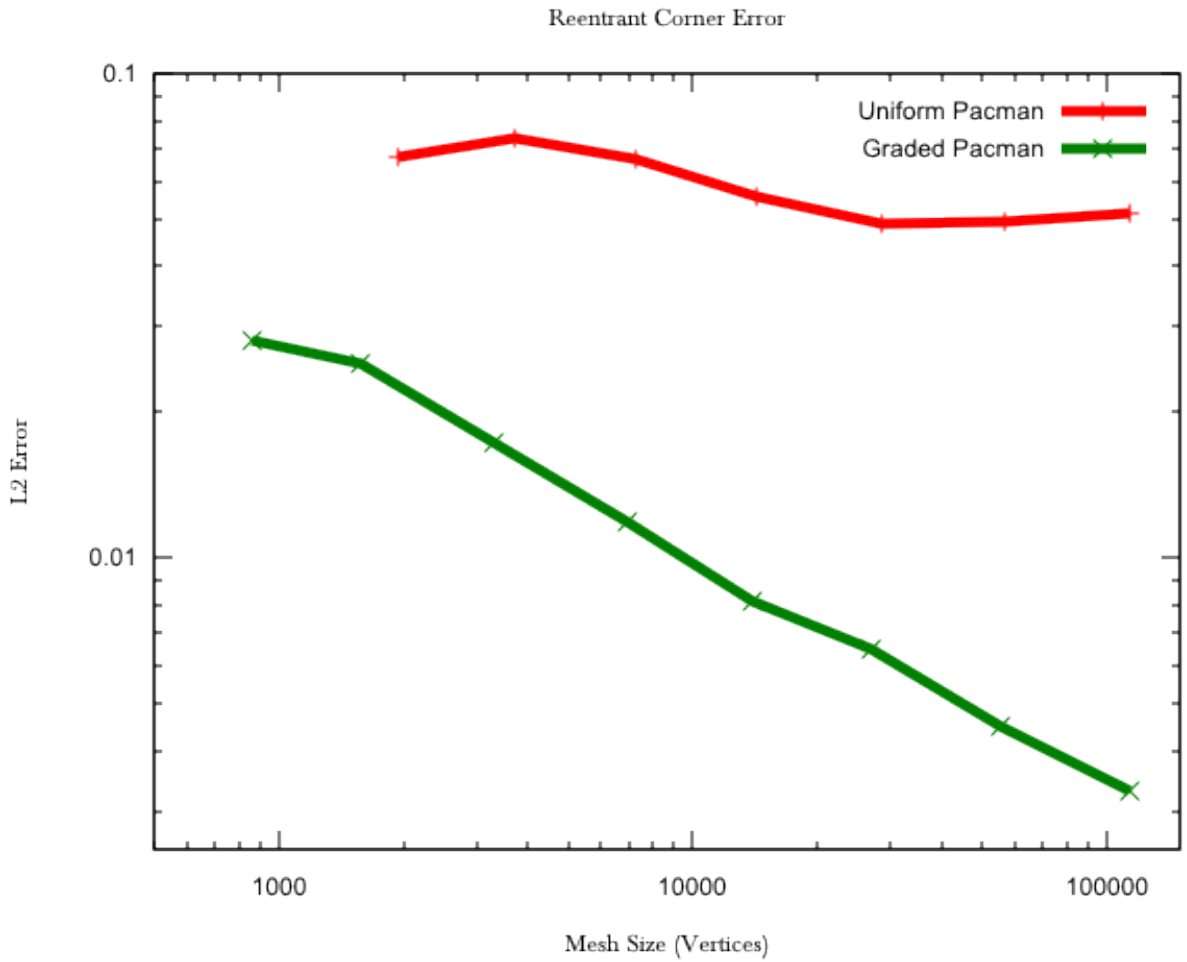
$$C_{low} r^{1-\mu} \leq h \leq C_{high} r^{1-\mu}$$

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



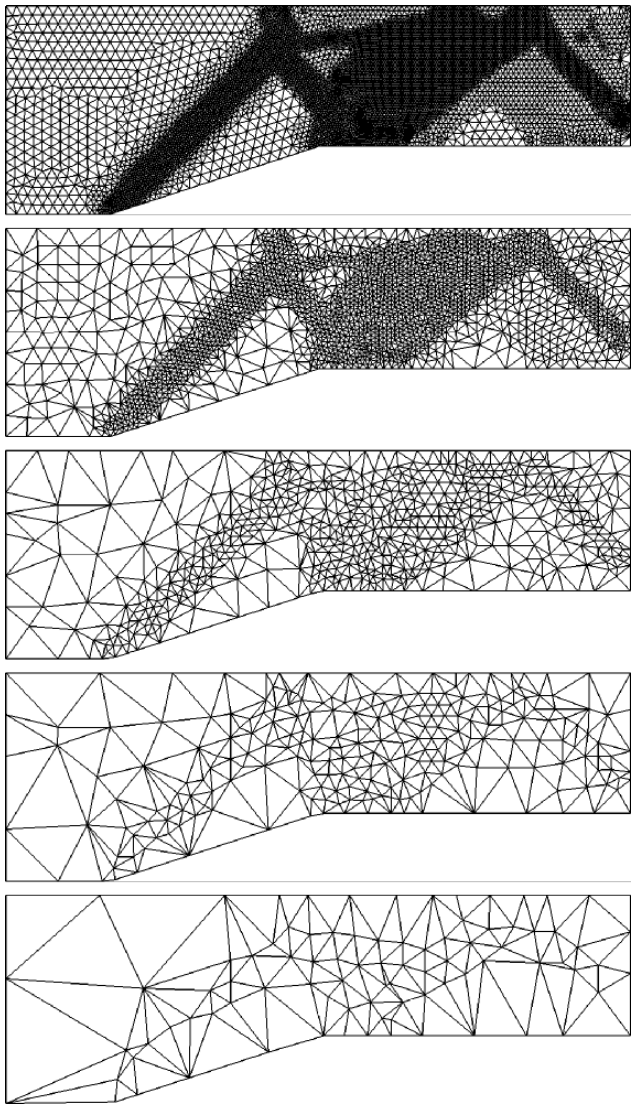
The Folly of Uniform Refinement

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

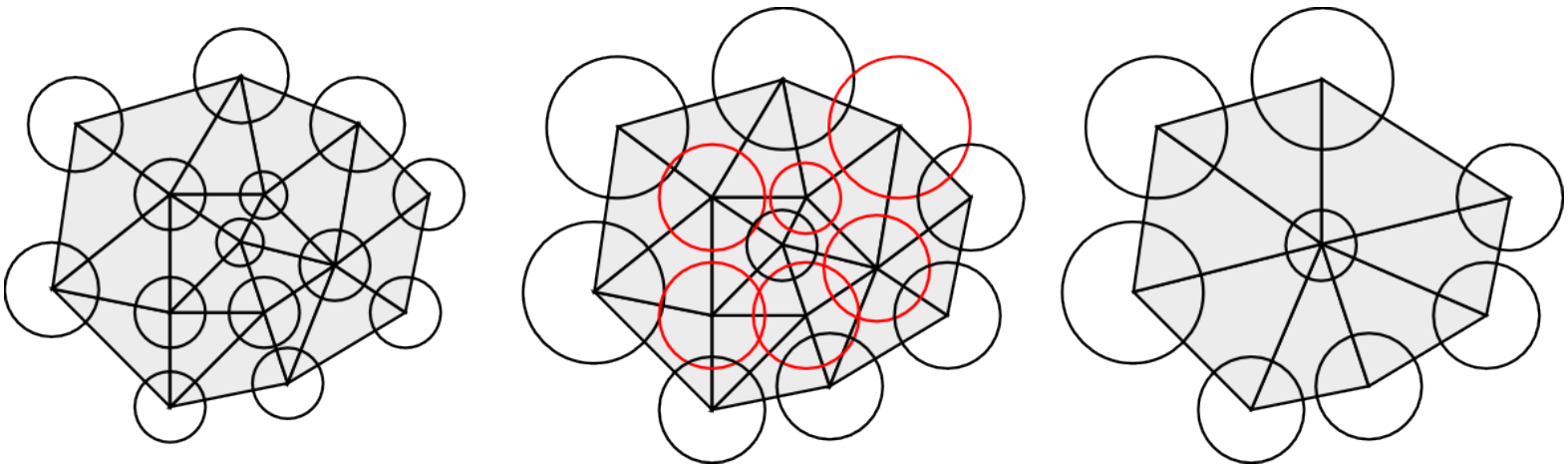
- Sufficient conditions for optimal-order convergence:
  - $|M_c| < 2|M_f|$ in terms of cells
  - any cell in M_c overlaps a bounded # of cells in M_f
  - monotonic increase in cell length-scale
- Each M_k satisfies the **quasi-uniformity** condition:

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

  - 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

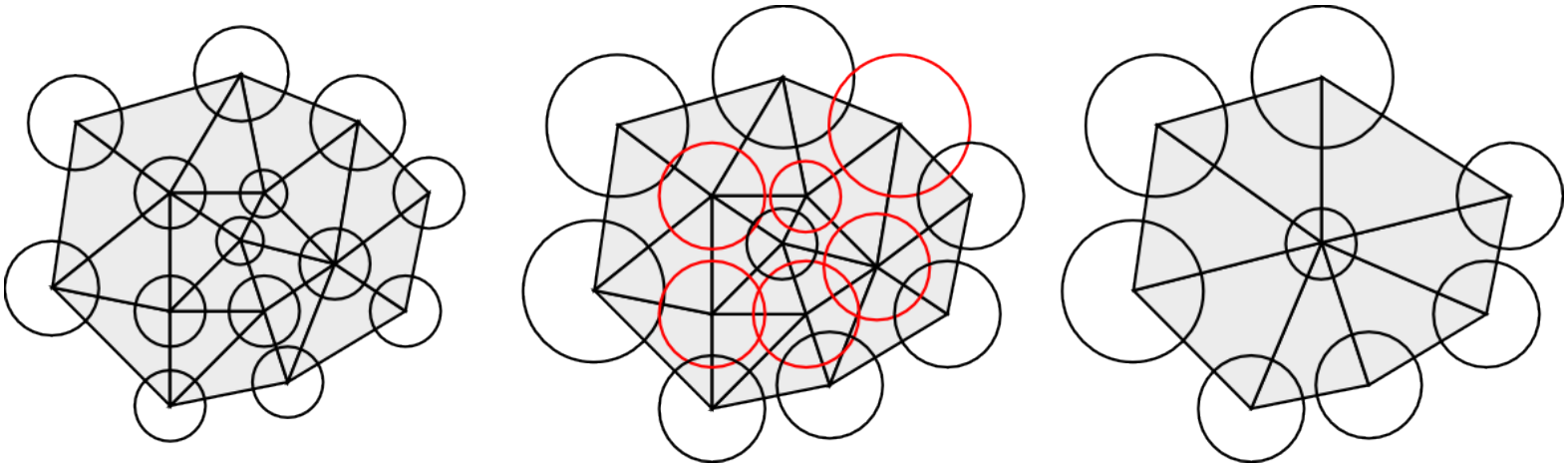
Function Based Coarsening

- (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

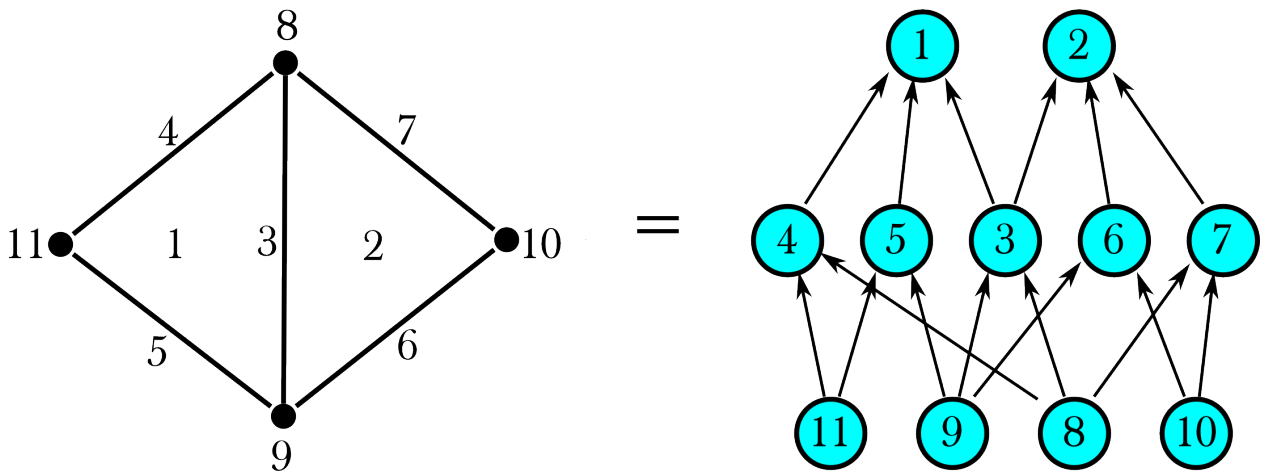
Decimation Algorithm

- 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

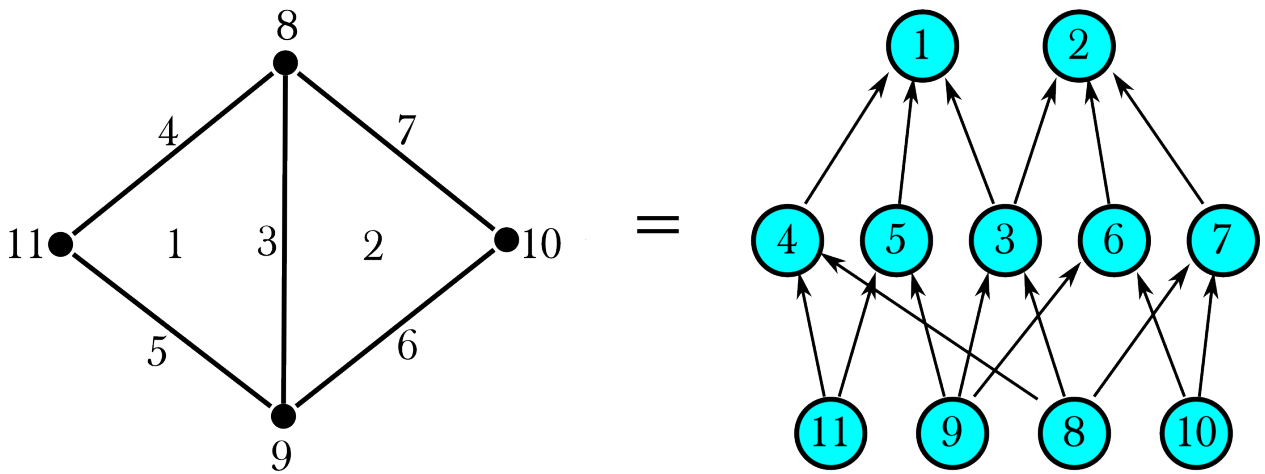
- vertex neighbors: $cone(support(v)) \setminus v$
- vertex link: $closure(star(v)) \setminus star(closure(v))$
- connectivity graph induced by limiting sieve depth
- remeshing can be handled as local modifications on the sieve
- meshing operations, such as *cone construction* easy



Implementation in *Sieve*

Peter Brune, 2008

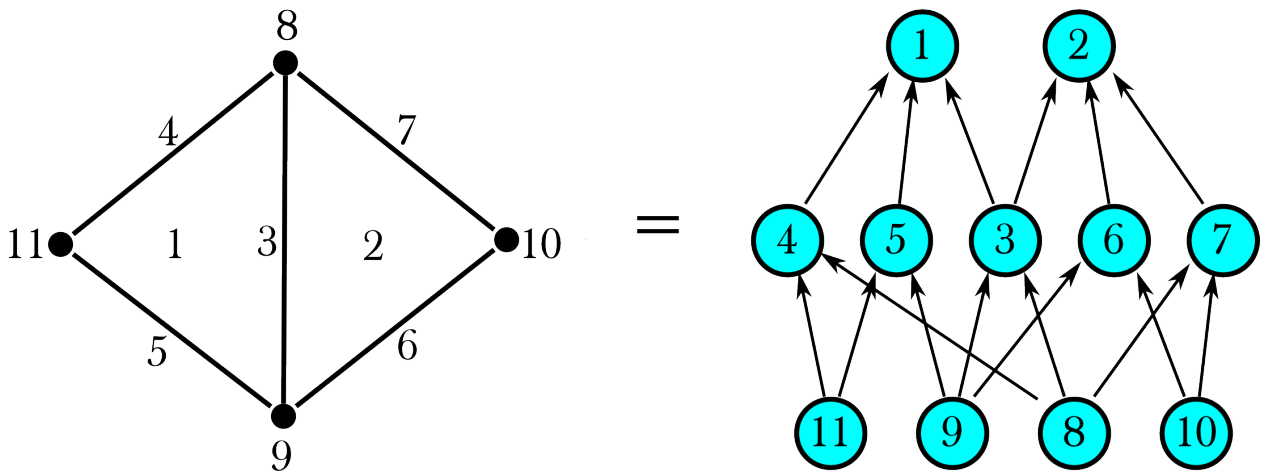
- vertex neighbors: $cone(support(v)) \setminus v$
- vertex link: $closure(star(v)) \setminus star(closure(v))$
- connectivity graph induced by limiting sieve depth
- remeshing can be handled as local modifications on the sieve
- meshing operations, such as *cone construction* easy



Implementation in *Sieve*

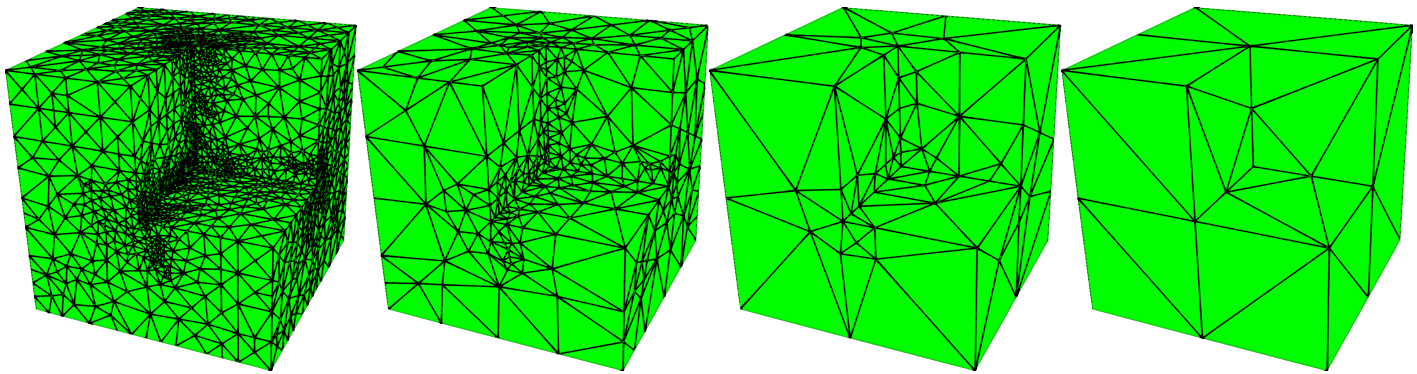
Peter Brune, 2008

- 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



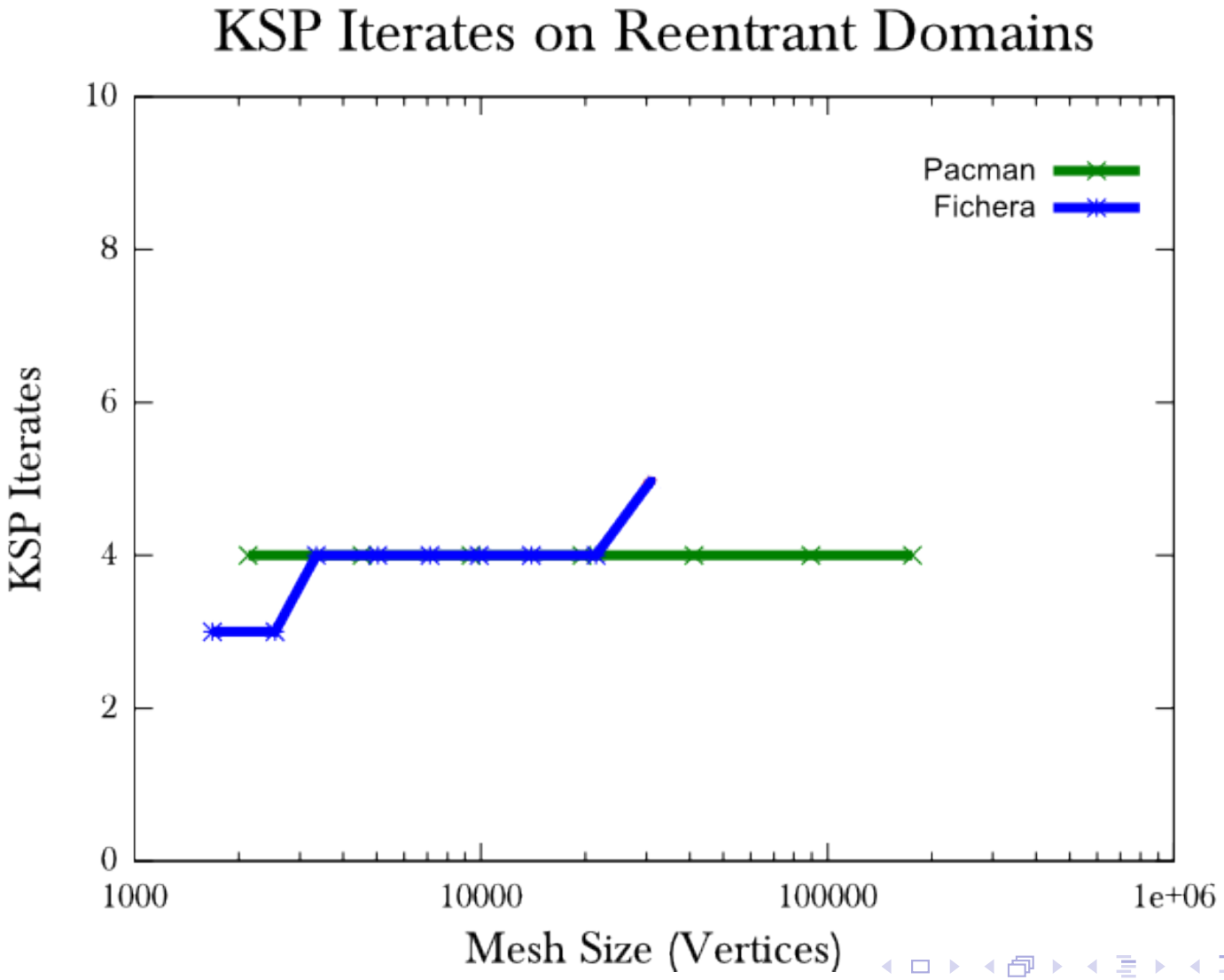
3D Test Problem

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



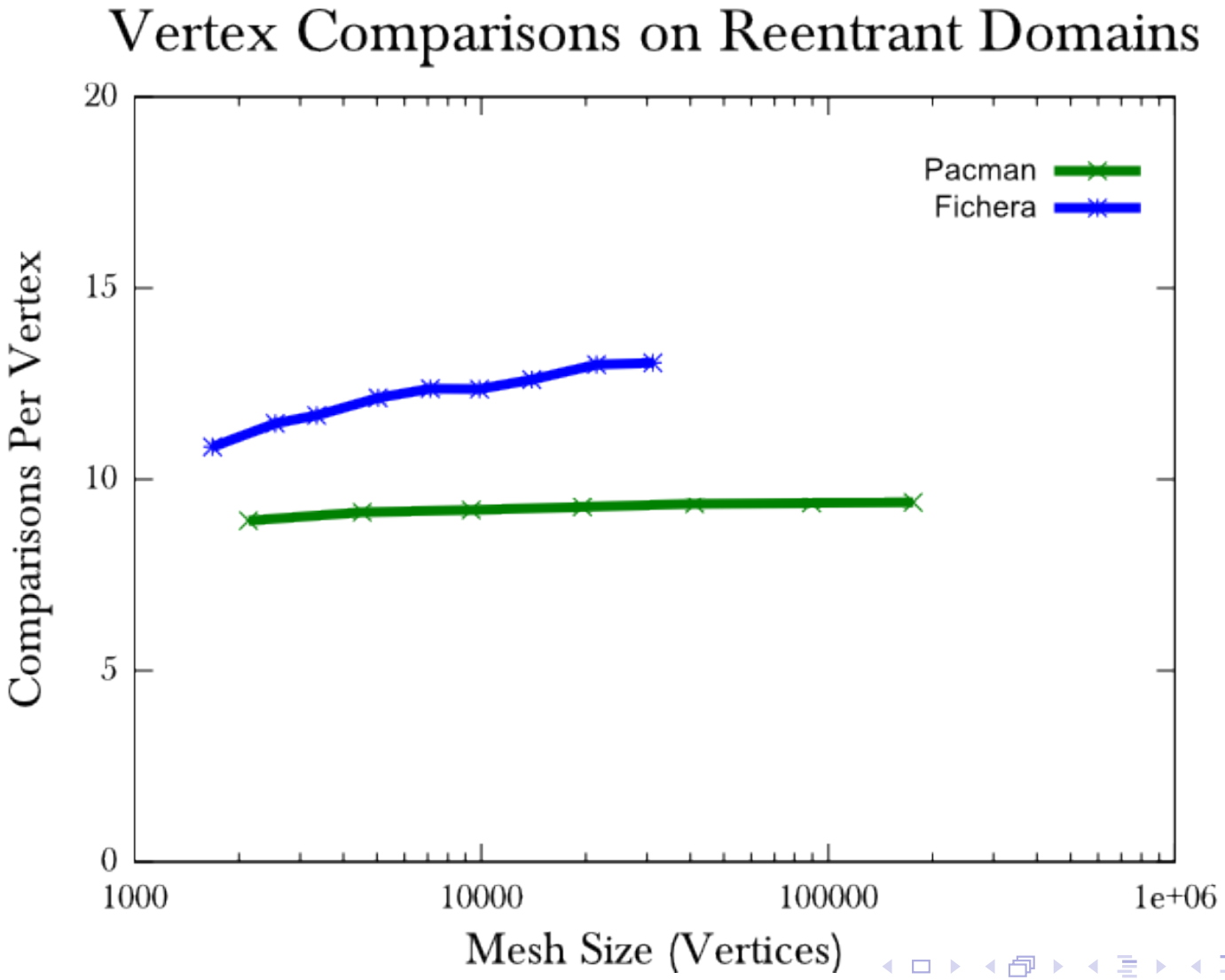
GMG Performance

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

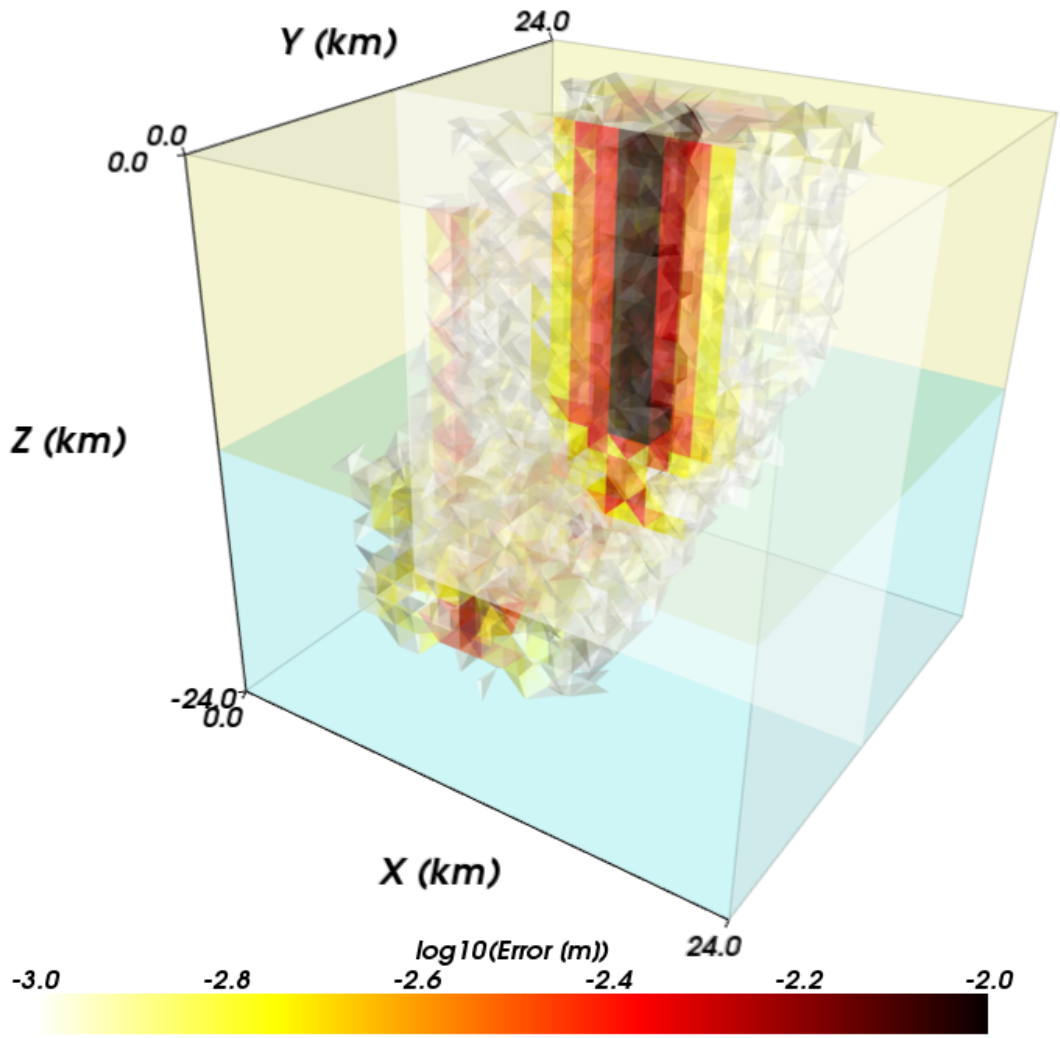
Pacman Mesh, $\beta = 1.45$						
level	cells	vertices	$\frac{min(h_K)}{h_k}$	max $\frac{h_K}{\rho_k}$	min(h_K)	max. overlap
0	19927	10149	0.020451	4.134135	0.001305	-
1	5297	2731	0.016971	4.435928	0.002094	23
2	3028	1572	0.014506	4.295703	0.002603	14
3	1628	856	0.014797	5.295322	0.003339	14
4	863	464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

Outline

- FEM
- UMG
- PyLith

PyLith

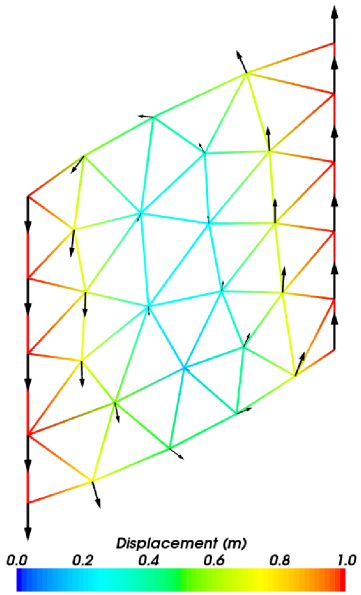
- Multiple problems
  - Dynamic rupture
  - Quasi-static relaxation
- Multiple models
  - Nonlinear visco-plastic
  - Finite deformation
  - Fault constitutive models
- Multiple meshes
  - 1D, 2D, 3D
  - Hex and tet meshes
- Parallel
  - PETSc solvers
  - DMPlex mesh management

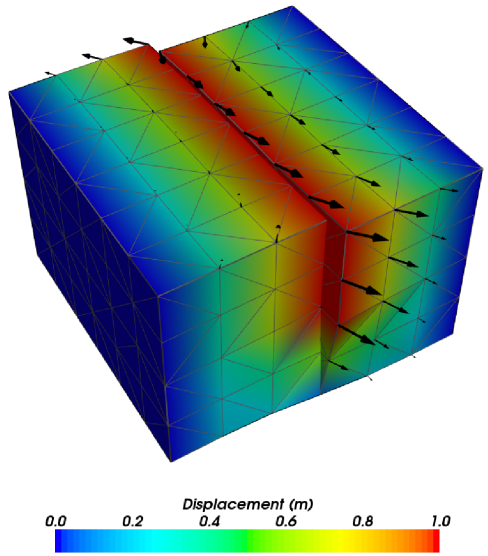


[a] Aagaard, Knepley, Williams

Multiple Mesh Types

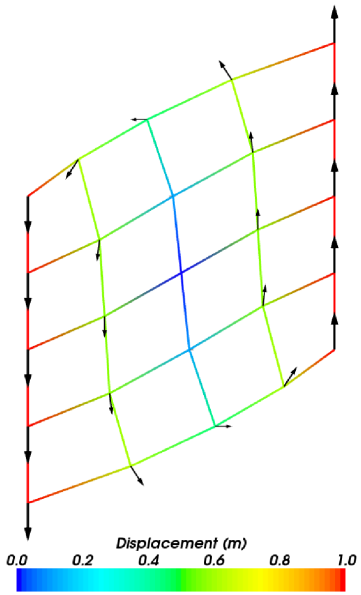
Triangular

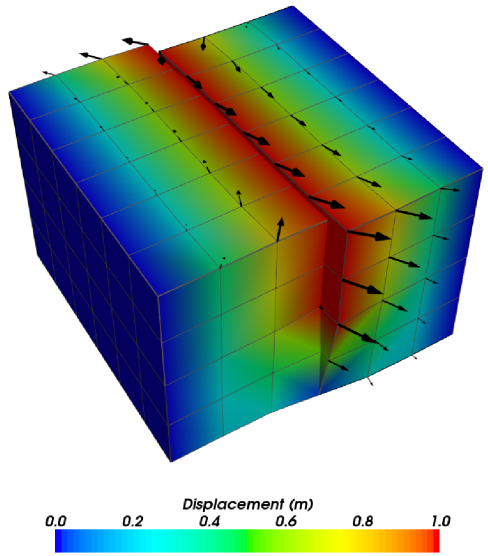




Tetrahedral

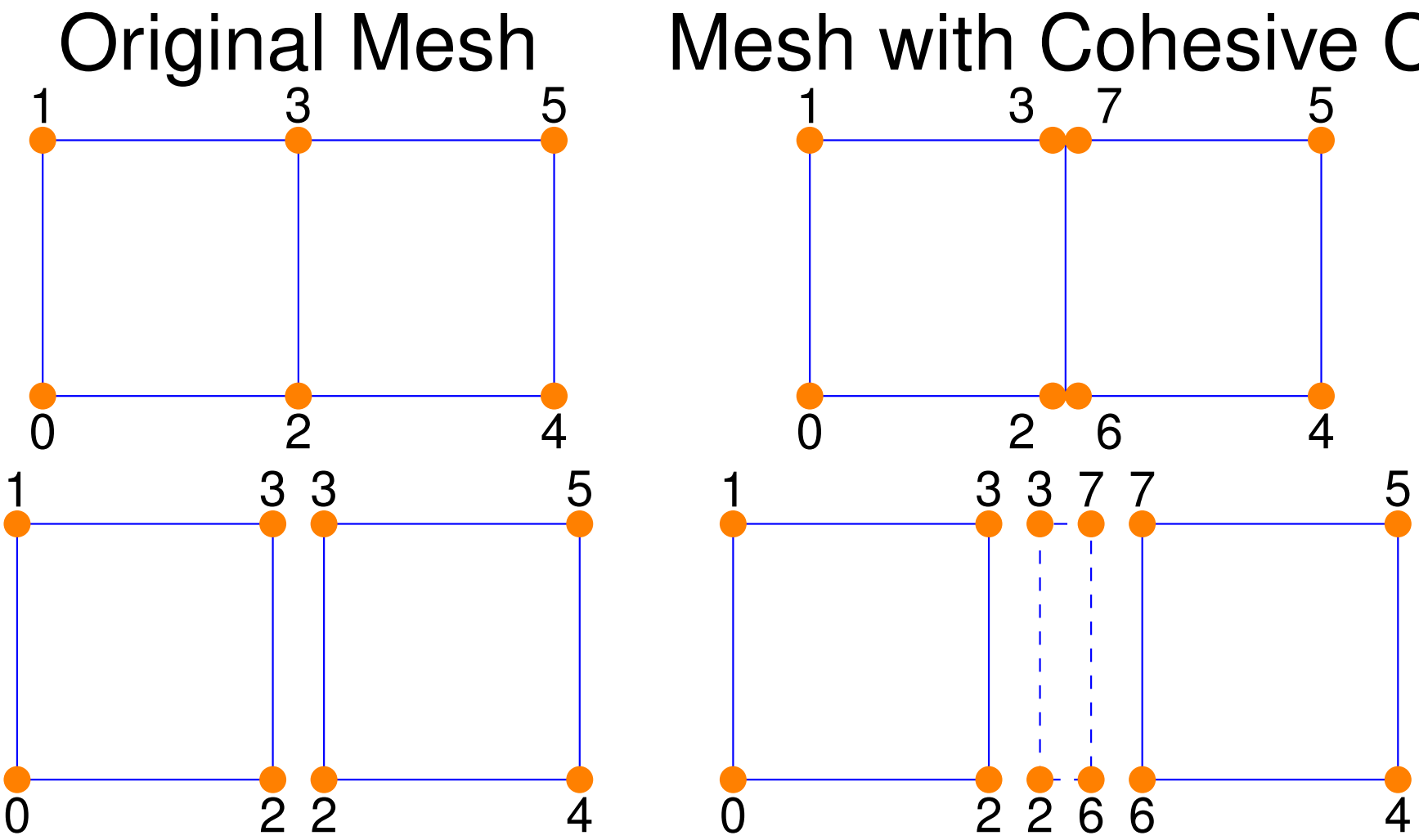
Rectangular





Hexahedral

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

Reverse-slip Benchmark