Implementation for Scientific Computing: FEM and FMM

Matthew Knepley

Computation Institute
University of Chicago

Department of Mathematics
Tufts University
March 12, 2010
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
Automatically Optimize

- Loop Tiling
- FErari
- PetFMM
How?

Simplify Design

- Generic type systems
- Sieve
- PetFMM
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

1. Introduction

2. Operator Assembly
   - Problem Statement
   - Plan of Attack
   - Results
   - Mixed Integer Linear Programming

3. Mesh Distribution

4. Parallel FMM
A familiar problem, FEM assembly, is recast to allow automatic optimization.
A familiar problem, FEM assembly, is recast to allow automatic optimization.
Main Point

A familiar problem, FEM assembly, is recast to allow automatic optimization.
Operator Assembly

Problem Statement

Plan of Attack

Results

Mixed Integer Linear Programming
Form Decomposition

Element integrals are decomposed into **analytic** and **geometric** parts:

\[
\int_T \nabla \phi_i(x) \cdot \nabla \phi_j(x) \, dx \tag{1}
\]

\[
= \int_T \frac{\partial \phi_i(x)}{\partial x_\alpha} \frac{\partial \phi_j(x)}{\partial x_\alpha} \, dx \tag{2}
\]

\[
= \int_{T_{\text{ref}}} \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} |J|\, dx \tag{3}
\]

\[
= \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \xi_\gamma}{\partial x_\alpha} |J| \int_{T_{\text{ref}}} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} \, dx \tag{4}
\]

\[
= G^{\beta\gamma}(T) K_{\beta\gamma}^{ij} \tag{5}
\]

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 $G(T)$

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>0</th>
<th>0</th>
<th>-1</th>
<th>1</th>
<th>1</th>
<th>-4</th>
<th>-4</th>
<th>0</th>
<th>4</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-4</td>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>-4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-4</td>
<td>-4</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>-4</td>
<td>-8</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>-4</td>
<td>8</td>
<td>8</td>
<td>-8</td>
<td>-4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>-4</td>
<td>-8</td>
<td>4</td>
<td>8</td>
<td>-4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>-4</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-4</td>
<td>0</td>
<td>-4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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 $\text{flops}(v^Tg)$ 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

2 Operator Assembly
- Problem Statement
- Plan of Attack
- Results
- Mixed Integer Linear Programming
If $v_j^T g$ is known, is $\text{flops}(v_j^T g) < 2m - 1$?

We can use binary relations among the vectors:

- **Equality**
  - If $v_j = v_i$, then $\text{flops}(v_j^T g) = 0$

- **Colinearity**
  - If $v_j = \alpha v_i$, then $\text{flops}(v_j^T g) = 1$

- **Hamming distance**
  - If $\text{dist}_H(v_j, v_i) = k$, then $\text{flops}(v_j^T g) = 2k$
Algorithm for Binary Relations

- Construct a weighted graph on $v_i$
  - The weight $w(i,j)$ is $\text{flops}(v_j^T g)$ 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 $\text{flops}(v_k^T g) = 3$
  - 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

2. Operator Assembly
   - Problem Statement
   - Plan of Attack
   - Results
   - Mixed Integer Linear Programming
Finite Element rearrangement to automatically reduce instructions

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

<table>
<thead>
<tr>
<th>Order</th>
<th>Entries</th>
<th>Base MAPs</th>
<th>FErari MAPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>24</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>84</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>55</td>
<td>220</td>
<td>45</td>
</tr>
<tr>
<td>4</td>
<td>120</td>
<td>480</td>
<td>176</td>
</tr>
<tr>
<td>5</td>
<td>231</td>
<td>924</td>
<td>443</td>
</tr>
<tr>
<td>6</td>
<td>406</td>
<td>1624</td>
<td>867</td>
</tr>
</tbody>
</table>
Outline

2. Operator Assembly
   - Problem Statement
   - Plan of Attack
   - Results
   - Mixed Integer Linear Programming
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

\[ \alpha_{ij} = \text{Basis expansion coefficients} \]
\[ y_i = \text{Binary variable indicating membership in the basis} \]
\[ s_{ij} = \text{Binary variable indicating nonzero coefficient } \alpha_{ij} \]
\[ z_{ij} = \text{Binary variable linearizes the objective function (equivalent to } y_i y_j) \]
\[ U = \text{Upper bound on coefficients} \]

Constraints

- \( Eq. (6b) \) : Basis expansion
- \( Eq. (6c) \) : Exclude nonbasis vector from the expansion
- \( Eq. (6d) \) : Remove offdiagonal coefficients for basis vectors
- \( Eq. (7c) \) : Exclude vanishing coefficients from cost
MINLP Model

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

subject to \[ v_i = \sum_{j=1}^{n} \alpha_{ij} v_j \] \hspace{1cm} i = 1, \ldots, n \hspace{1cm} (6b)

\[ -Uy_j \leq \alpha_{ij} \leq Uy_j \] \hspace{1cm} i, j = 1, \ldots, n \hspace{1cm} (6c)

\[ -U(1 - y_i) \leq \alpha_{ij} \leq U(1 - y_i) \] \hspace{1cm} i, j = 1, \ldots, n, \hspace{1cm} i \neq j \hspace{1cm} (6d)

\[ y_i \in \{0, 1\} \] \hspace{1cm} i = 1, \ldots, n \hspace{1cm} (6e)
Equivalent MILP Model: \( z_{ij} = y_i \cdot y_j \)

\[
\begin{align*}
\text{minimize} & \quad 2m \sum_{i=1}^{n} y_i + 2 \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} (y_j - z_{ij}) - n \\
\text{subject to} & \quad v_i = \sum_{j=1}^{n} \alpha_{ij} v_j & i = 1, \ldots, n \\
& \quad -Uy_j \leq \alpha_{ij} \leq Uy_j & i, j = 1, \ldots, n \\
& \quad -U(1 - y_i) \leq \alpha_{ij} \leq U(1 - y_i) & i, j = 1, \ldots, n, \ i \neq j \\
& \quad z_{ij} \leq y_i, \ z_{ij} \leq y_j, \ z_{ij} \geq y_i + y_j - 1, & i, j = 1, \ldots, n \\
& \quad y_i \in \{0, 1\}, \ z_{ij} \in \{0, 1\} & i, j = 1, \ldots, n.
\end{align*}
\]
Take advantage of sparsity of $\alpha_{ij}$ coefficient

Introduce binary variables $s_{ij}$ to model existence of $\alpha_{ij}$

Add constraints $-Us_{ij} \leq \alpha_{ij} \leq Us_{ij}$
Sparse Coefficient Formulation

MINLP Model

\[
\begin{align*}
\text{minimize} & \quad \sum_{i=1}^{n} \left\{ y_i (2m - 1) + (1 - y_i) \left( 2 \sum_{j=1, j \neq i}^{n} s_{ij} - 1 \right) \right\} \\
\text{subject to} & \quad v_i = \sum_{j=1}^{n} \alpha_{ij} v_j \\
& \quad -Us_{ij} \leq \alpha_{ij} \leq Us_{ij} \\
& \quad -U(1 - y_i) \leq \alpha_{ij} \leq U(1 - y_i) \\
& \quad s_{ij} \leq y_j \\
& \quad y_i \in \{0, 1\}, \quad s_{ij} \in \{0, 1\}
\end{align*}
\]
Sparse Coefficient Formulation

Equivalent MILP Model

\[
\begin{align*}
\text{minimize} & \quad 2m \sum_{i=1}^{n} y_i + 2 \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} (s_{ij} - z_{ij}) - n \\
\text{subject to} & \quad v_j = \sum_{j=1}^{n} \alpha_{ij} v_j \\
& \quad -Us_{ij} \leq \alpha_{ij} \leq Us_{ij} \\
& \quad -U(1 - y_i) \leq \alpha_{ij} \leq U(1 - y_i) \\
& \quad z_{ij} \leq y_i, \quad z_{ij} \leq s_{ij}, \quad z_{ij} \geq y_i + s_{ij} - 1, \\
& \quad y_i \in \{0, 1\}, \quad z_{ij} \in \{0, 1\}, \quad s_{ij} \in \{0, 1\}
\end{align*}
\]
Results

Initial Formulation

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

<table>
<thead>
<tr>
<th>Element</th>
<th>Default Flops</th>
<th>MILP Flops</th>
<th>MILP LPs</th>
<th>MILP CPU</th>
<th>Sparse Coef. MILP Flops</th>
<th>Sparse Coef. MILP LPs</th>
<th>Sparse Coef. MILP CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$ 2D</td>
<td>42</td>
<td>42</td>
<td>33</td>
<td>0.10</td>
<td>34</td>
<td>187</td>
<td>0.43</td>
</tr>
<tr>
<td>$P_2$ 2D</td>
<td>147</td>
<td>147</td>
<td>2577</td>
<td>37.12</td>
<td>67</td>
<td>6030501</td>
<td>36000</td>
</tr>
<tr>
<td>$P_1$ 3D</td>
<td>170</td>
<td>166</td>
<td>79</td>
<td>0.49</td>
<td>146</td>
<td>727</td>
<td>3.97</td>
</tr>
<tr>
<td>$P_2$ 3D</td>
<td>935</td>
<td>935</td>
<td>25283</td>
<td>36000</td>
<td>829</td>
<td>33200</td>
<td>36000</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Elements</th>
<th>Sparse Coefficient Flops</th>
<th>Sparse Basis Flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$ 2D</td>
<td>34</td>
<td>12</td>
</tr>
<tr>
<td>$P_1$ 3D</td>
<td>146</td>
<td>26</td>
</tr>
</tbody>
</table>
Outline

1. Introduction

2. Operator Assembly

3. Mesh Distribution
   - Sieve
   - Distribution
   - Interfaces
   - Full Assembly

4. Parallel FMM
Rethinking meshes
produces a simple FEM interface
and good code reuse.
Rethinking meshes produces a simple FEM interface and good code reuse.
Rethinking meshes produces a simple FEM interface and good code reuse.
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?”
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?”
Mesh Distribution

- Sieve
  - Distribution
  - Interfaces
  - Full Assembly
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
- $\text{cone}(0) = \{2, 3, 4\}$
- $\text{support}(7) = \{2, 3\}$
Incidence/covering arrows

closure(0) = \{0, 2, 3, 4, 7, 8, 9\}

star(7) = \{7, 2, 3, 0\}
Doublet Mesh

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

$\text{meet}(0, 1) = \{4\}$

$\text{join}(8, 9) = \{4\}$
Doublet Mesh

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

Construct mesh dual by reversing sieve arrows, taking the support of each face, and taking the meet of each cell pair.
**Doublet Section**

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

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

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

- $\text{restrict}(0) = \{f_0\}$
- $\text{restrict}(2) = \{v_0\}$
- $\text{restrict}(6) = \{e_0, e_1\}$
**Topological traversals: follow connectivity**

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

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

- \textit{restrictClosure}(0) = \{ f_0 e_0 e_1 e_2 e_3 e_4 e_5 v_0 v_1 v_2 \}
- \textit{restrictStar}(7) = \{ v_0 e_0 e_4 e_5 f_0 \}
Localization
- Restrict to patches (here an edge closure)
- Compute locally
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 local patch data
- Completion = restrict → fuse → update, in parallel
A ubiquitous parallel form of restrict $\rightarrow$ fuse $\rightarrow$ update

Operates on Sections
- Sieves can be "downcast" to Sections

Based on two operations
- Data exchange through overlap
- Fusion of shared data
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
  - accumulating matvec for a partially assembled matrix
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
  - accumulating matvec for a partially assembled matrix
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
  - accumulating matvec for a partially assembled matrix
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
  - accumulating matvec for a partially assembled matrix
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
  - accumulating matvec for a partially assembled matrix
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
  - accumulating matvec for a partially assembled matrix
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
  - accumulating matvec for a partially assembled matrix
3 Mesh Distribution

- Sieve
- Distribution
- Interfaces
- Full Assembly
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)!
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 $\text{cone}(\cdot)$'s!
Mesh Partition

- 3rd party packages construct a vertex partition
- For FEM, partition dual graph vertices
- For FVM, construct hypergraph dual with faces as vertices
- Assign $\text{closure}(v)$ and $\text{star}(v)$ to same partition
Doublet Mesh Distribution

Process 0

Process 1

PT
Doublet Mesh Distribution

Mesh Distribution

process 0

process 1

0,1 0,2 1,1 1,2 0,3 0,4 0,5 1,3 1,4 0,6 0,5

0,3 0,4 0,5 1,3 1,4

0,2 1,1 1,2 1,3 1,4 0,5 0,6

process 0

process 1

PT

M. Knepley (UC)
Doublet Mesh Distribution

process 0

process 1

P

T

process 0

process 1
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

M. Knepley (UC)
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
A simple triangular mesh
Sieve for the mesh
Local sieve on process 0
2D Example

Partition Overlap

Process 0

Process 1
2D Example

Partition Section

3  5  6  7  11  12  13  14  15  16
Updated Sieve Overlap
2D Example

Cone Section

Mesh Distribution

Distribution
2D Example

Distributed Sieve

Process 0

Process 1
2D Example

Coordinate Section

```
| 0.5 | 0.0
|-----|--
| 0.0 | 0.5
| 0.0 | 0.0
| 0.5 | 0.5
| 1.0 | 0.0
| 1.0 | 0.5
| 0.0 | 1.0
| 0.5 | 1.0
| 1.0 | 1.0
```

Nodes: 8, 9, 10, 11, 12, 13, 14, 15, 16
2D Example

Distributed Coordinate Section

Process 0

Process 1
Distributed Mesh
3D Example

A simple hexahedral mesh
3D Example

Sieve for the mesh

It's complicated!
3D Example

Sieve for the mesh

It's complicated!
3D Example

Partition Overlap

Process 0

Process 1
3D Example

Partition Section

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>24</td>
<td>25</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>34</td>
</tr>
<tr>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
</tr>
<tr>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
</tr>
<tr>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>89</td>
<td>96</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>107</td>
<td>108</td>
</tr>
</tbody>
</table>
3D Example

Distributed Mesh

Notice cells are ghosted
Outline

3 Mesh Distribution
- Sieve
- Distribution
- Interfaces
- Full Assembly
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

<table>
<thead>
<tr>
<th>Local (analytical)</th>
<th>Global (topological)</th>
</tr>
</thead>
</table>
| - Discretization/Approximation  
  - FEM integrals  
  - FV fluxes |
| - Boundary conditions |
| - Largely dim dependent (e.g. quadrature) |
| - 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
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
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
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

Mesh Distribution
- Sieve
- Distribution
- Interfaces
- Full Assembly
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 */
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 */
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 */
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 */
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 */
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 */
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 */
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 */
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];
        }
        /* Nonlinear term */
        elemVec[f] *= weight[q] * detJ;
    }
    /* Update output vector */
}
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 */
}
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 */
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 */
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 */
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 */
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

```c
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

\[\text{aAagaard, Knepley, Williams}\]
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
  - Backtracking

- No prescribed cracks (movie)
  - Arbitrary crack geometry
  - Arbitrary intersections

- Multiple materials
  - Composite toughness

^aBourdin

\(^a\)M. Knepley (UC)
Fracture Mechanics

Bourdin

---

1 Bourdin
Outline

1. Introduction
2. Operator Assembly
3. Mesh Distribution
4. Parallel FMM
   - Short Introduction to FMM
   - Parallelism
   - PetFMM
Using estimates and proofs, a simple software architecture achieves good scaling and adaptive load balance.
Using estimates and proofs, a simple software architecture, achieves good scaling and adaptive load balance.
Using estimates and proofs, a simple software architecture, achieves good scaling and adaptive load balance.
Outline

4 Parallel FMM
- Short Introduction to FMM
  - Parallelism
  - PetFMM
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
- $O(N)$ time
- Distributed and multicore (GPU) parallelism
- Small memory bandwidth requirement
FMM accelerates the calculation of the function:

\[ \Phi(x_i) = \sum_j K(x_i, x_j)q(x_j) \]  

(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
Parallel FMM
Short Introduction to FMM

Fast Multipole Method

FMM accelerates the calculation of the function:

\[ \Phi(x_i) = \sum_j \frac{q_j}{|x_i - x_j|} \]  

- Accelerates \( O(N^2) \) to \( O(N) \) 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.
Kernel operations will map to GPU tasks.
Kernel operations will map to GPU tasks.
Outline

4 Parallel FMM
- Short Introduction to FMM
- Parallelism
- PetFMM
Parallel FMM

FMM in Sieve

The Quadtree is a Sieve with optimized operations

- Multiploes 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 Quadtrees 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
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 Quadtrees 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
Kernel operations will map to GPU tasks.
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: \( N_i p (P2M) + n_i p^2 (M2L) + N_i p (L2P) + 3^d N_i^2 (P2P) \)
    - 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 dimension \( m \)
- Leverage existing work on graph partitioning
  - ParMetis

- Good partitions exist for non-uniform distributions
  - 2D $O(\sqrt{n}(\log n)^{3/2})$ edgecut
  - 3D $O(n^{2/3}(\log n)^{4/3})$ 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?


- 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 average 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

- Uniform 4ML8R5
- Uniform 10ML9R5
- Spiral 1ML8R5
- Spiral w/space-filling 1ML8R5

Number of processors vs. load balance for different load balancing strategies.

- M. Knepley (UC)
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

4 Parallel FMM
   - Short Introduction to FMM
   - Parallelism
   - 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

- uniform 4ML8R5
- uniform 10ML9R5
- spiral 1ML8R5
- spiral w/ space-filling 1ML8R5
- Perfect Speedup
PetFMM CPU Performance

Strong Scaling

![Graph showing performance metrics for PetFMM CPU with strong scaling. The graph plots time in seconds against the number of processors. Key performance metrics include ME Initialization, Upward Sweep, Downward Sweep, Evaluation, Load balancing stage, and Total time. The y-axis is logarithmic, ranging from $10^{-2}$ to $10^3$, and the x-axis ranges from 2 to 256 processors. The data points indicate performance improvement as the number of processors increases.]
Better mathematical abstractions bring concrete benefits

- Vast reduction in complexity
  - Dimension and mesh independent code
  - Complete serial code reuse

- Opportunities 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
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 is a compiler for variational forms by Anders Logg.

Here is a mixed-form Poisson equation:

\[ a((\tau, w), (\sigma, u)) = L((\tau, w)) \quad \forall (\tau, w) \in V \]

where

\[ a((\tau, w), (\sigma, u)) = \int_{\Omega} \tau\sigma - \nabla \cdot \tau u + w \nabla \cdot u \, dx \]

\[ L((\tau, w)) = \int_{\Omega} wf \, dx \]
```python
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
```
Here is a discontinuous Galerkin formulation of the Poisson equation:

\[ a(v, u) = L(v) \quad \forall v \in V \]

where

\[
a(v, u) = \int_{\Omega} \nabla u \cdot \nabla v \, dx \\
+ \sum_{S} \int_{S} - \langle \nabla v \rangle \cdot [u]_{n} - [v]_{n} \cdot \langle \nabla u \rangle - (\alpha/h)vu \, dS \\
+ \int_{\partial\Omega} -\nabla v \cdot [u]_{n} - [v]_{n} \cdot \nabla u - (\gamma/h)vu \, ds \\
L(v) = \int_{\Omega} vf \, dx
\]
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
For the Poisson problem, meshes with reentrant corners have a length-scale requirement in order to maintain accuracy:

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

\[ \mu \leq \frac{\pi}{\theta} \]
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 $M_k$
  - $\rho_K$ 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 $M_k$
  - $\rho_K$ 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: $S(v) = \frac{\text{dist}(NN(v), v)}{2}$
Function Based Coarsening

- pick a subset of the vertices such that $\beta(S(v) + S(w)) > \text{dist}(v, w)$
- 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.

Even 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
vertex neighbors: \( \text{cone}(\text{support}(v)) \setminus v \)
vertex link: \( \text{closure}(\text{star}(v)) \setminus \text{star}(\text{closure}(v)) \)

connectivity graph induced by limiting sieve depth
remeshing can be handled as local modifications on the sieve
meshing operations, such as cone construction easy
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
vertex neighbors: \( \text{cone}(\text{support}(v)) \setminus v \)

vertex link: \( \text{closure}(\text{star}(v)) \setminus \text{star}(\text{closure}(v)) \)

connectivity graph induced by limiting sieve depth

remeshing can be handled as local modifications on the sieve

meshing operations, such as \textit{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:

![Graph showing KSP iterates on reentrant domains](image-url)
GMG Performance

Coarsening work is nearly constant as system size increases:

Vertex Comparisons on Reentrant Domains

Comparison Per Vertex vs Mesh Size (Vertices)

- Pacman (green)
- Fichera (blue)
## Quality Experiments

Table: Hierarchy quality metrics - 2D

<table>
<thead>
<tr>
<th>level</th>
<th>cells</th>
<th>vertices</th>
<th>$\frac{\min(h_K)}{h_k}$</th>
<th>$\max \frac{h_k}{\rho_k}$</th>
<th>$\min(h_K)$</th>
<th>max. overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19927</td>
<td>10149</td>
<td>0.020451</td>
<td>4.134135</td>
<td>0.001305</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5297</td>
<td>2731</td>
<td>0.016971</td>
<td>4.435928</td>
<td>0.002094</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>3028</td>
<td>1572</td>
<td>0.014506</td>
<td>4.295703</td>
<td>0.002603</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>1628</td>
<td>856</td>
<td>0.014797</td>
<td>5.295322</td>
<td>0.003339</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>863</td>
<td>464</td>
<td>0.011375</td>
<td>6.403574</td>
<td>0.003339</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>449</td>
<td>250</td>
<td>0.022317</td>
<td>6.330512</td>
<td>0.007979</td>
<td>13</td>
</tr>
</tbody>
</table>
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
  - DMPllex mesh management

\[\text{a Aagaard, Knepley, Williams}\]
Multiple Mesh Types

- Triangular
- Tetrahedral
- Rectangular
- Hexahedral
Cohesive Cells

Original Mesh

Mesh with Cohesive Cell

Exploded view of meshes
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
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
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
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
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
Further Work

PyLith

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
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

Further Work

PyLith

M. Knepley (UC)