Tree-based methods on GPUs

Felipe Cruz[1] and Matthew Knepley[2,3]

[1]Department of Mathematics
University of Bristol

[2]Computation Institute
University of Chicago

[3]Department of Molecular Biology and Physiology
Rush University Medical Center

SIAM CS & E
Miami, FL Mar 2, 2009

RUSH UNIVERSITY MEDICAL CENTER

Outline

Scientific Computing Challenge

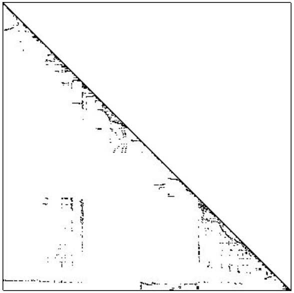
How do we create
reusable
implementations which are also
efficient?

Scientific Computing Insight

Structures are conserved,

but tradeoffs change.

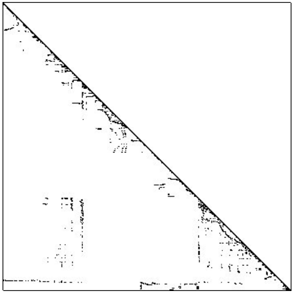
Structure vs. Tradeoffs



This is how PETSc works:

- Sparse matrix-vector product has a common structure
- Different storage formats are chosen based upon
  - architecture
  - PDE

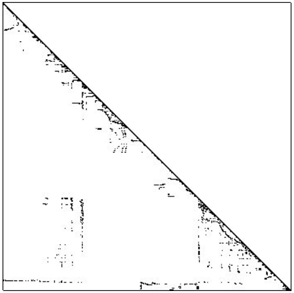
Structure vs. Tradeoffs



This is how PETSc works:

- Sparse matrix-vector product has a common structure
- Different storage formats are chosen based upon
  - architecture
  - PDE

Structure vs. Tradeoffs



This is how PETSc works:

- Sparse matrix-vector product has a common structure
- Different storage formats are chosen based upon
  - architecture
  - PDE

Structure vs. Tradeoffs

$$\mathbf{A\ x = b}$$

$$\mathbf{\{\ b,\ Ab,\ A(Ab),\ A(A(Ab)),\ \ldots\ \}}$$

This is how PETSc works:

- Krylov solvers have a common structure
- Different solvers are chosen based upon
  - problem characteristics
  - architecture

Structure vs. Tradeoffs

$$\mathbf{A\,x = b}$$

$$\mathbf{\{\, b,\ Ab,\ A(Ab),\ A(A(Ab)),\ \ldots \,\}}$$

This is how PETSc works:

- Krylov solvers have a common structure
- Different solvers are chosen based upon
  - problem characteristics
  - architecture

Structure vs. Tradeoffs

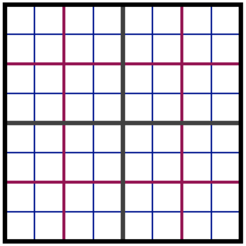
$$\mathbf{A\,x = b}$$

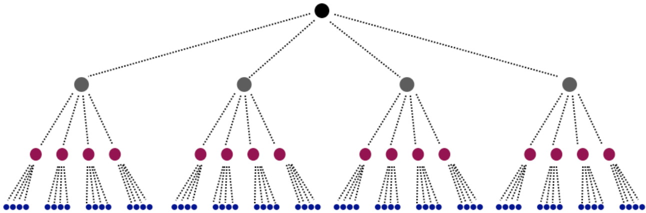
$$\mathbf{\{\, b, Ab, A(Ab), A(A(Ab)), \ldots \,\}}$$

This is how PETSc works:

- Krylov solvers have a common structure
- Different solvers are chosen based upon
  - problem characteristics
  - architecture

Structure vs. Tradeoffs

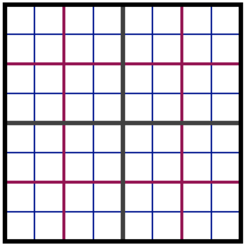


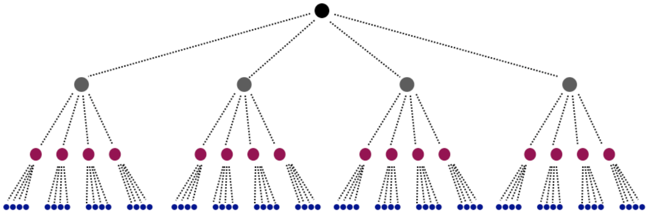


This is how treecodes work:

- Hierarchical algorithms have a common structure
- Different analytical and geometric decisions depend upon
  - problem configuration
  - accuray requirements

Structure vs. Tradeoffs

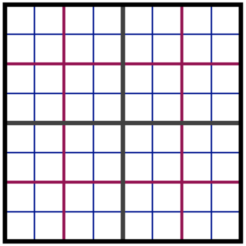


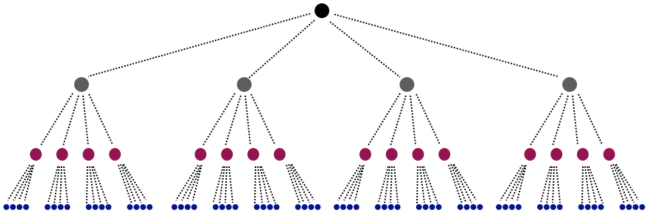


This is how treecodes work:

- Hierarchical algorithms have a common structure
- Different analytical and geometric decisions depend upon
    - problem configuration
    - accuray requirements

Structure vs. Tradeoffs

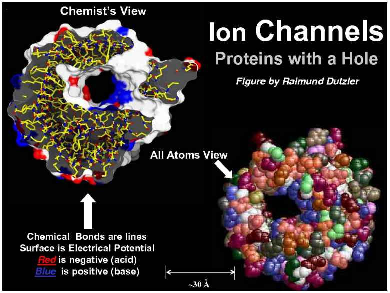




This is how treecodes work:

- Hierarchical algorithms have a common structure
- Different analytical and geometric decisions depend upon
    - problem configuration
    - accuray requirements

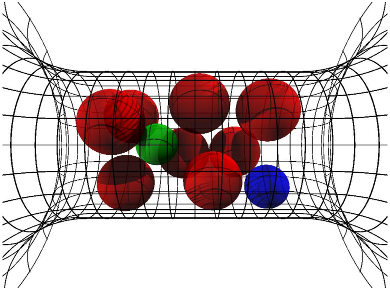
Structure vs. Tradeoffs



This is how biology works:

- For ion channels, Nature uses the same
  - protein building blocks
  - energetic balances
- Different energy terms predominate for different uses

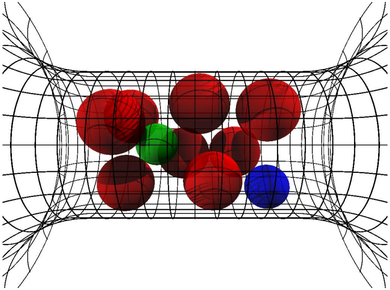
Structure vs. Tradeoffs



This is how biology works:

- For ion channels, Nature uses the same
  - protein building blocks
  - energetic balances
- Different energy terms predominate for different uses

Structure vs. Tradeoffs



This is how biology works:

- For ion channels, Nature uses the same
  - protein building blocks
  - energetic balances
- Different energy terms predominate for different uses

Representation Hierarchy

Divide the work into levels:

- Model
- Algorithm
- Implementation

Representation Hierarchy

Divide the work into levels:

- Model
- Algorithm
- Implementation

Spiral Project:

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

Representation Hierarchy

Divide the work into levels:

- Model
- Algorithm
- Implementation

FLAME Project:

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

Representation Hierarchy

Divide the work into levels:

- Model
- Algorithm
- Implementation

FEniCS Project:

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

Representation Hierarchy

Divide the work into levels:

- Model
- Algorithm
- Implementation

Treecodes:

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

Representation Hierarchy

Divide the work into levels:

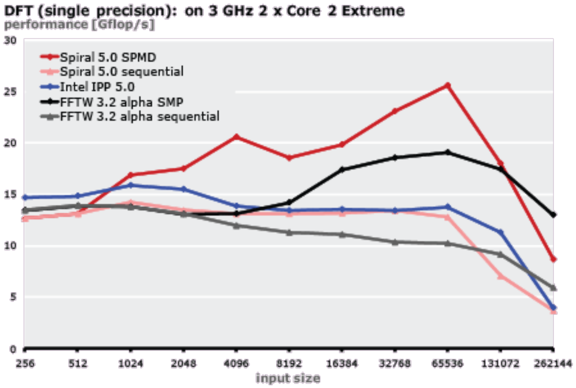
- Model
- Algorithm
- Implementation

Treecodes:

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

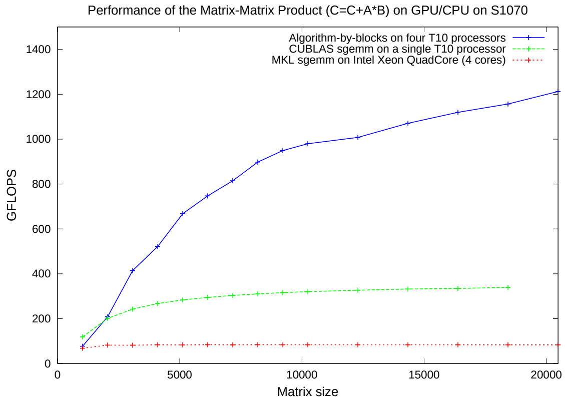
Each level demands a strong abstraction layer

Spiral



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

FLAME & FLASH

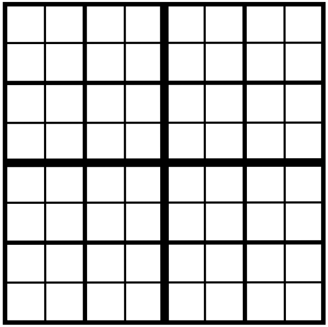


- Robert van de Geijn, http://www.cs.utexas.edu/users/flame
- FLAME is an Algorithm-By-Blocks interface
- FLASH/SuperMatrix is a runtime system

Outline

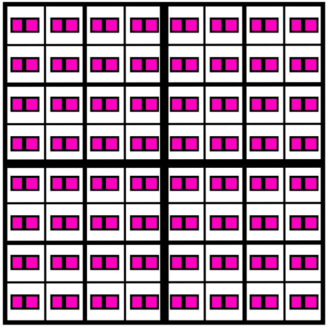
Outline

FMM in Sieve



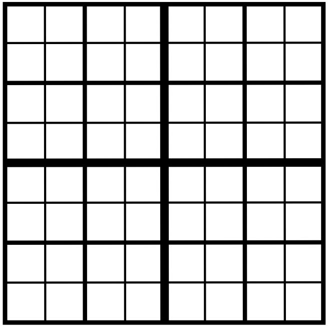
- The Quadtree is a `Sieve`
    - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
    - Neighbors
    - Interaction List
- Completion moves data for
    - Neighbors
    - Interaction List

FMM in Sieve



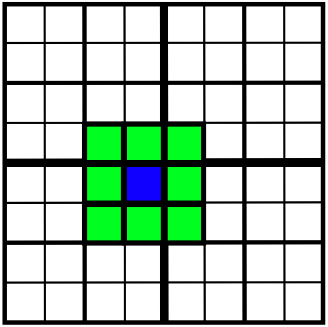
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



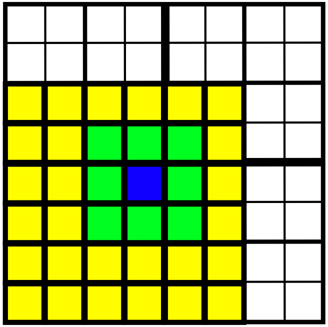
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



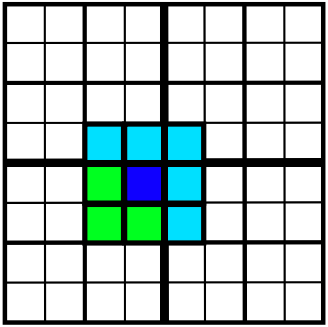
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



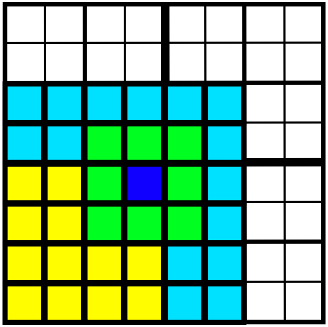
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



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

FMM in Sieve



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

Tree Interface

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

Outline

FMM Sections

FMM requires data over the Quadtree distributed by:

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

FMM Sections

FMM requires data over the Quadtree distributed by:

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

FMM Sections

FMM requires data over the Quadtree distributed by:

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

FMM Sections

FMM requires data over the Quadtree distributed by:

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

Notice this is multiscale since data is divided at each level

Outline

Evaluator Interface

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

Evaluator Interface

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

Kernel Interface

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

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

Outline

Greengard & Gropp Analysis

For a shared memory machine,

$$T = a\frac{N}{P} + b\log_4 P + c\frac{N}{BP} + d\frac{NB}{P} + e(N,P) \qquad (1)$$

1. Initialize multipole expansions, finest local expansions, final sum
2. Reduction bottleneck
3. Translation and Multipole-to-Local
4. Direct interaction
5. Low order terms

A Parallel Version of the Fast Multipole Method,
L. Greengard and W.D. Gropp, *Comp. Math. Appl.*, **20**(7), 1990.

Distributed FMM

Additions for distributed computing:

- Partitioning
- Explicit optimization problem to minimize
  - Communication volume
  - Load imbalance
- Uses PETSc Sieve for parallelism

Distributed FMM

Additions for distributed computing:

- Partitioning
- Explicit optimization problem to minimize
  - Communication volume
  - Load imbalance
- Uses PETSc Sieve for parallelism

Distributed FMM

Additions for distributed computing:

- Partitioning
- Explicit optimization problem to minimize
  - Communication volume
  - Load imbalance
- Uses PETSc Sieve for parallelism

Distributed FMM

Additions for distributed computing:

- Partitioning
- Explicit optimization problem to minimize
  - Communication volume
  - Load imbalance
- Uses PETSc Sieve for parallelism

Outline

Question

What is the optimal number of particles per cell?

- Greengard & Gropp
    - Minimize time and maximize parallel efficiency
    - $B_{opt} = \sqrt{\frac{c}{d}} \approx 30$
- Gumerov & Duraiswami
    - Follow GG, but also try to consider memory access
    - $B_{opt} \approx 91$, but instead, they choose 320
    - Heavily weights the N^2 part of the computation
- We propose to cover up the bottleneck with direct evaluations

Problem
Missing Concurrency

We can balance time in direct evaluation with idle time for small grids.

- The direct evaluation takes time $d\frac{NB}{p}$
- Assume a single thread group works on the first L tree levels

Thus, we need

$$B \geq \frac{b}{d}\frac{4^{L+1}p}{N} \tag{2}$$

in order to cover the bottleneck. In an upcoming publication, we show that this bound holds for all modern processors.

Problem

Missing Bandwidth

We can restructure the M2L to conserve bandwidth

- Matrix-free application of M2L
- Reorganize traversal to minimize bandwidth

Old Pull in 27 interaction MEs, transform to LE, reduce

New Pull in cell ME, transform to 27 interaction LEs, partially reduce

Matrix-Free M2L

The M2L transformation applies the operator

$$M_{ij} = -1^i t^{-(i+j+1)} \binom{i+j}{j} \tag{3}$$

Notice that the *t* exponent is constant along perdiagonals. Thus we

- divide by *t* at each perdiagonal
- calculate the C_{ij} by the recurrence along each perdiagonal
- carefully formulate complex division (STL fails here)

Outline

Outline

FLASH Design

FLASH enables multicore computing through FLAME

- LA interface is identical to FLAME
- FLAME executes operates immediately
- FLASH queues operations, and
- Executes queues on user call (does nothing in FLAME)

FLASH Design

FLASH enables multicore computing through FLAME

- LA interface is identical to FLAME
- FLAME executes operates immediately
- FLASH queues operations, and
- Executes queues on user call (does nothing in FLAME)

FLASH Design

FLASH enables multicore computing through FLAME

- LA interface is identical to FLAME
- FLAME executes operates immediately
- FLASH queues operations, and
- Executes queues on user call (does nothing in FLAME)

FLASH Design

FLASH enables multicore computing through FLAME

- LA interface is identical to FLAME
- FLAME executes operates immediately
- FLASH queues operations, and
- Executes queues on user call (does nothing in FLAME)

Cholesky Factorization

```
FLA_Part_2x2(A, &ATL, &ATR,
                &ABL, &ABR, 0, 0, FLA_TL);
while(FLA_Object_length(ATL) < FLA_Object_length(A)) {
  FLA_Repart_2x2_to_3x3(
    ATL, ATR, &A00, &A01, &A02,
              &A10, &A11, &A12,
    ABL, ABR, &A20, &A21, &A22, 1, 1, FLA_BR);
  FLASH_Chol(FLA_UPPER_TRIANGULAR, A11);
  FLASH_Trsm(FLA_LEFT,FLA_UPPER_TRIANGULAR,FLA_TRANSPOSE,
             FLA_NONUNIT_DIAG, FLA_ONE, A11, A12);
  FLASH_Syrk(FLA_UPPER_TRIANGULAR, FLA_TRANSPOSE,
             FLA_MINUS_ONE, A12, FLA_ONE, A22);
  FLA_Cont_with_3x3_to_2x2(
    &ATL, &ATR, A00, A01, A02,
                A10, A11, A12,
    &ABL, &ABR, A20, A21, A22, FLA_TL);
}
FLA_Queue_exec();
```

Outline

PetFMM-GPU

We break down sweep operations into `Tasks`

- Cell loops are now tiled
- Tasks are queued
- We can form a DAG since we know the dependence structure
- Scheduling is possible

This asynchronous interface can enable

- Overlapping direct and multipole calculations
- Reorganizing the downward sweep
- Adaptive expansions

GPU Classes

`Section`

- `size()` returns the number of values
- `getFiberDimension(cell)` returns the number of cell values
- `restrict/update()` retrieves and changes cell values
- `clone/extract()` converts between CPU and GPU objects

`Evaluator`

- `initializeExpansions()`
- `upwardSweep()`
- `downwardSweepTransform()`
- `downwardSweepTranslate()`
- `evaluateBlobs()`
- `evaluate()`

GPU Classes

`Section`

- `size()` returns the number of values
- `getFiberDimension(cell)` returns the number of cell values
- `restrict/update()` retrieves and changes cell values
- `clone/extract()` converts between CPU and GPU objects

`Task`

- Input data size
- Output data size
- Dependencies (future)

`TaskQueue`

- Manages storage and offsets
- `evaluate()`

Tasks

`Upward Sweep Task`

- cell block

in cell and child centers, child multipole coeff

out cell multipole coeff

`Downward Sweep Transform Task`

- cell block

in cell and interaction list centers, interaction list multipole coeff

out cell temp local coeff

`Downward Sweep Expansion Task`

- cell block

in cell and parent centers, cell temp local coeff, parent local coeff

out cell local coeff

Tasks

`Upward Sweep Task`

- cell block

in cell and child centers, child multipole coeff

out cell multipole coeff

`Downward Sweep Transform Task`

- cell block

in cell and interaction list centers, cell multipole coeff

out interaction list temp local coefficients

`Downward Sweep Expansion Task`

- cell block

in cell and parent centers, cell temp local coeff, parent local coeff

out cell local coeff

Tasks

`Upward Sweep Task`

- cell block

in cell and child centers, child multipole coeff

out cell multipole coeff

`Downward Sweep Reduce Task`

- cell block

in interaction list temp local coefficients

out cell temp local coefficients

`Downward Sweep Expansion Task`

- cell block

in cell and parent centers, cell temp local coeff, parent local coeff

out cell local coeff

Transform Task

Shifts interaction cell multipole expansion to cell local expansion

- Add a task for each interaction cell
- All tasks with same origin are merged
- Local memory:
  - 2 (p+1) blockSize (Pascal) + 2 p blockSize (LE) + 2 p (ME)

8 terms	4416 bytes
17 terms	9096 bytes

- Execution
  - 1 block per ME
  - Each thread reads a section of ME and the MEcenter
  - Each thread computes an LE separately
  - Each thread writes LE to separate global location

Reduce Task

Add up local expansion contributions from each interaction cell

- Add a task for each cell
- Local memory:
    - 2*terms (LE)

8 terms	64 bytes
17 terms	136 bytes

- Execution
    - 1 block per output LE
    - Each thread reads a section of input LE
    - Each thread adds to shared output LE

What's Important?

Interface improvements bring concrete benefits

- Facilitated code reuse
  - Serial code was largely reused
  - Test infrastructure completely reused
- Opportunites for performance improvement
  - Overlapping computations
  - Better task scheduling
- Expansion of capabilities
  - Could now combine distributed and multicore implementations
  - Could replace local expansions with cheaper alternatives

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) + $n_I p^2$ (M2L) + $n_c p^2$ (L2L)

  - Communication estimate:

Diagonal	$n_c(L - k - 1)$
Lateral	$2^d \frac{2^{m(L-k-1)}-1}{2^m-1}$ for incidence dimesion m

- Leverage existing work on graph partitioning
  - ParMetis

Parallel Tree Implementation
Why should a good partition exist?

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

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

Parallel Tree Implementation
Will ParMetis find it?

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

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

Parallel Tree Implementation

Advantages

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

Parallel Tree Implementation
Advantages

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

Parallel Tree Implementation
Advantages

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

Parallel Tree Interface

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

Parallel Data Movement

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

Parallel Evaluator Interface

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