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

School of Mathematical Sciences, Monash University
Clayton, VIC Mar 3, 2010

Outline

1. Introduction
2. Short Introduction to FMM
3. Serial Implementation
4. Parallel FMM
5. Multicore FMM

Scientific Computing Challenge

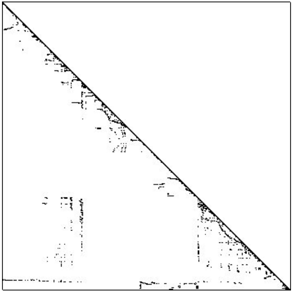
How do we create
reusable
implementations which are also
efficient?

Scientific Computing Insight

Structures are conserved,

but tradeoffs change.

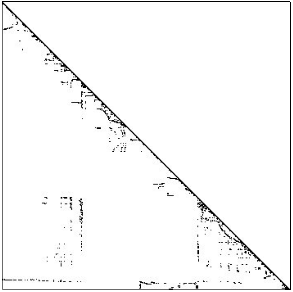
Structure vs. Tradeoffs



This is how PETSc works:

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

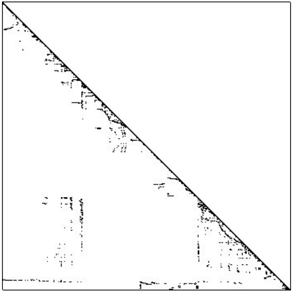
Structure vs. Tradeoffs



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

$$A\,x = b$$

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

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

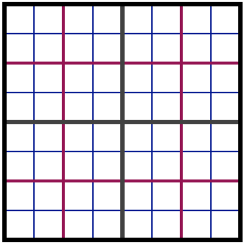
$$\mathbf{A\,x = b}$$

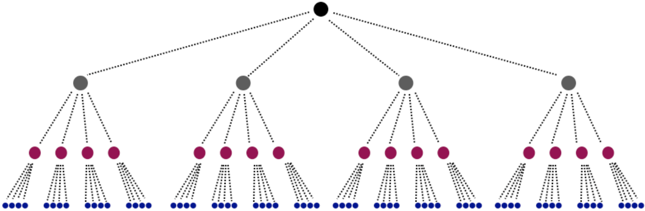
$$\mathbf{\{\,b,\ Ab,\ A(Ab),\ A(A(Ab)),\ \ldots\,\}}$$

This is how PETSc works:

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

Structure vs. Tradeoffs

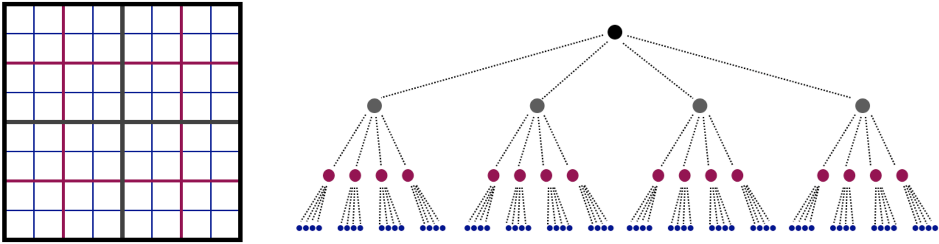




This is how treecodes work:

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

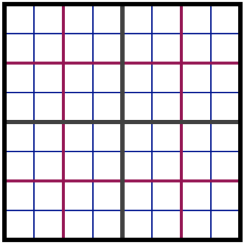
Structure vs. Tradeoffs

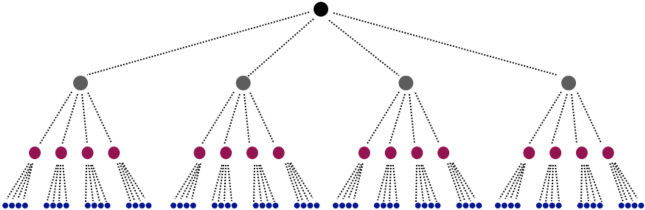


This is how treecodes work:

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

Structure vs. Tradeoffs

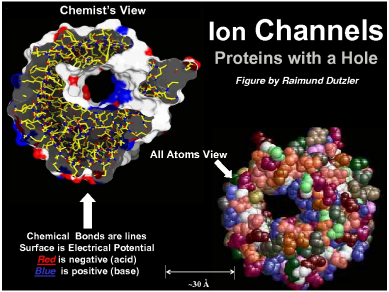




This is how treecodes work:

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

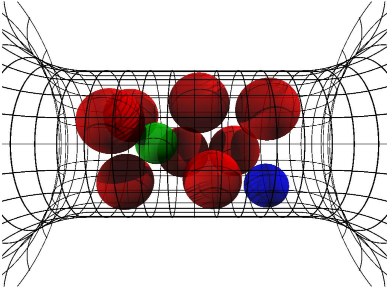
Structure vs. Tradeoffs



This is how biology works:

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

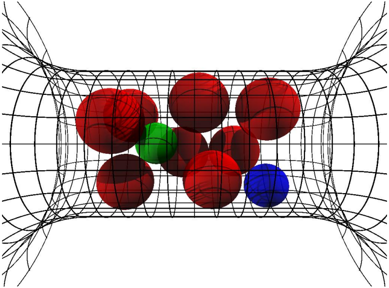
Structure vs. Tradeoffs



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:

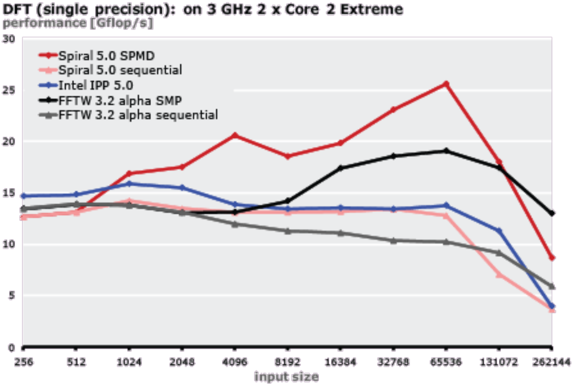
- Model
- Algorithm
- Implementation

Treecodes:

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

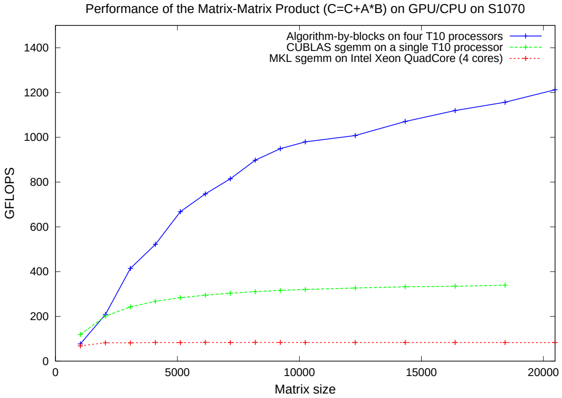
Each level demands a strong abstraction layer

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

1. Introduction
2. Short Introduction to FMM
   - Spatial Decomposition
   - Data Decomposition
3. Serial Implementation
4. Parallel FMM
5. Multicore FMM

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:

$$\Phi(x_i) = \sum_j K(x_i, x_j) q(x_j) \quad (1)$$

- 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:

$$\Phi(x_i) = \sum_j \frac{q_j}{|x_i - x_j|} \tag{1}$$

- 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

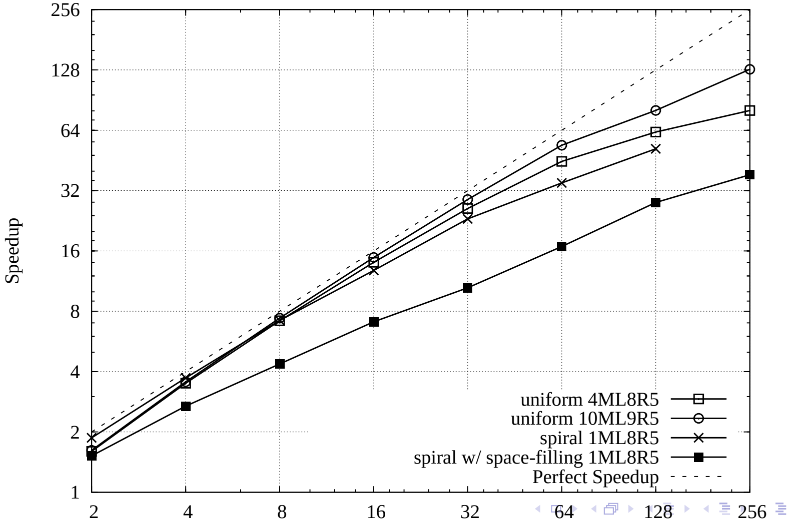
PetFMM

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

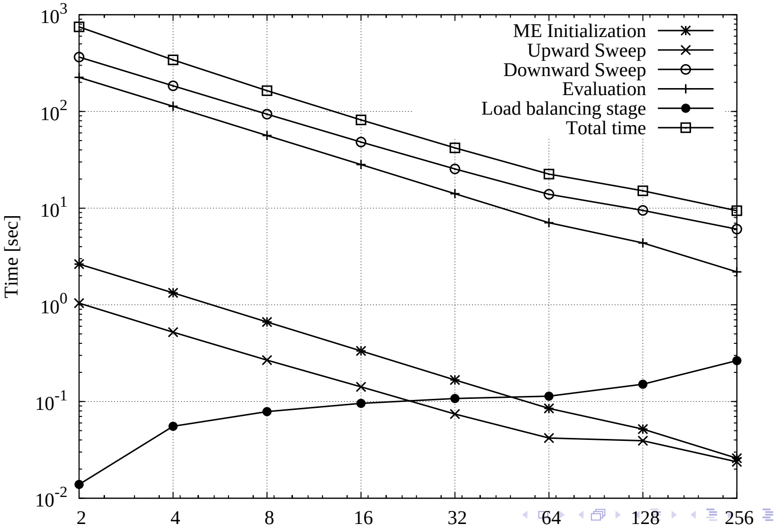
PetFMM CPU Performance

Strong Scaling



PetFMM CPU Performance

Strong Scaling

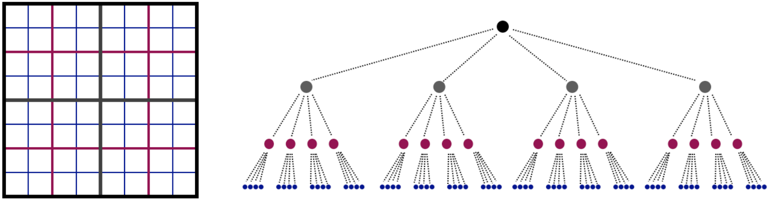


Outline

2 Short Introduction to FMM
- Spatial Decomposition
- Data Decomposition

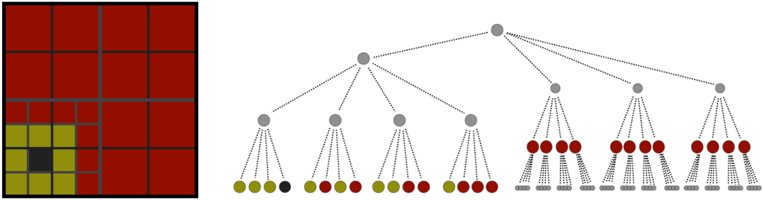
Spatial Decomposition

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



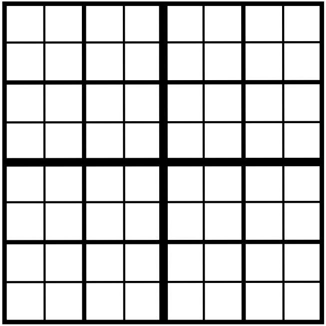
Spatial Decomposition

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



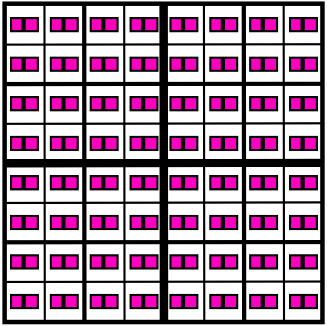
Neighbors are treated as *very near*.

FMM in Sieve



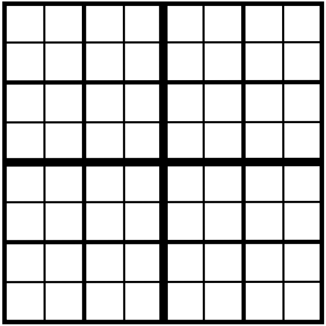
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



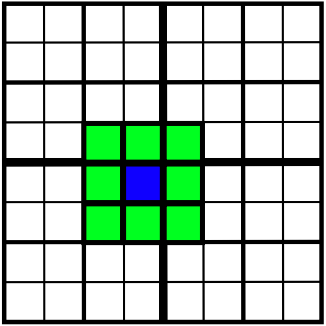
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



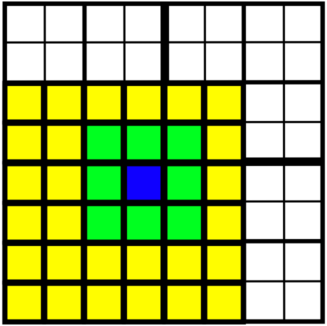
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



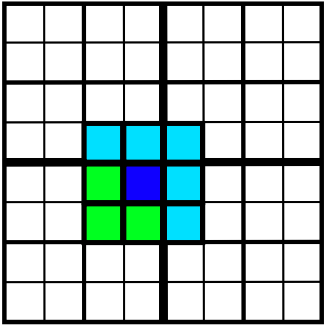
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

FMM in Sieve



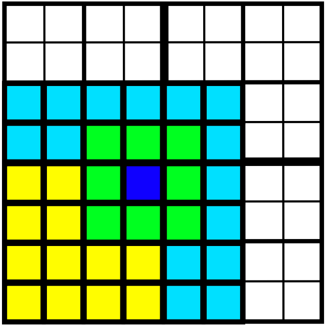
- The Quadtree is a `Sieve`
  - with optimized operations
- Multipoles are stored in `Sections`
- Two `Overlaps` are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List

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

Outline

2 Short Introduction to FMM
- Spatial Decomposition
- Data Decomposition

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

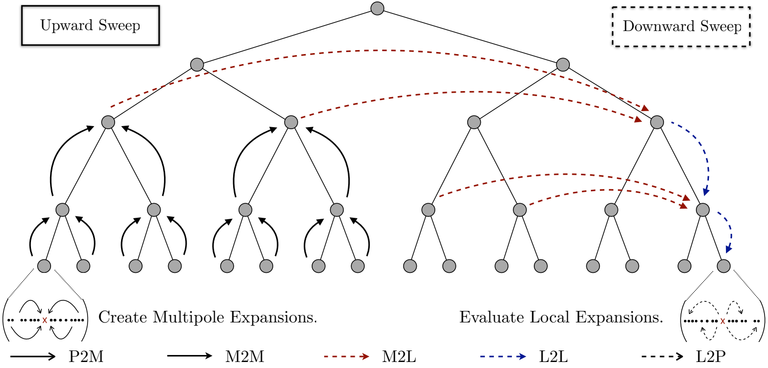
Outline

1. Introduction
2. Short Introduction to FMM
3. Serial Implementation
    - Control Flow
    - Interface
4. Parallel FMM
5. Multicore FMM

Outline

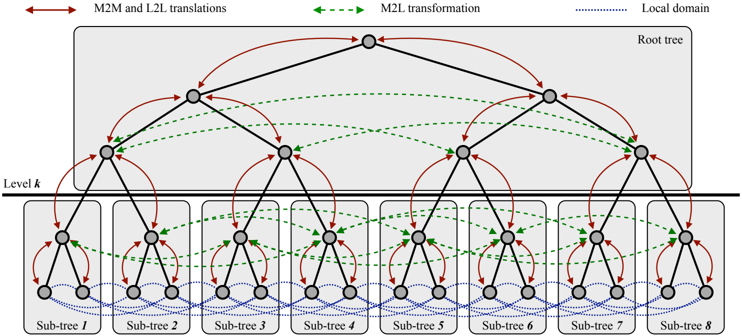
3. Serial Implementation
   - Control Flow
   - Interface

FMM Control Flow



Kernel operations will map to GPU tasks.

FMM Control Flow
Parallel Operation



Kernel operations will map to GPU tasks.

Outline

3 Serial Implementation
- Control Flow
- Interface

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

1. Introduction
2. Short Introduction to FMM
3. Serial Implementation
4. Parallel FMM
5. Multicore FMM

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

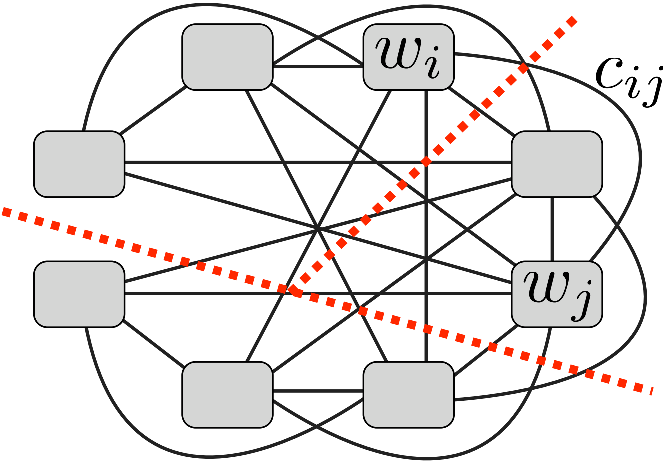
Parallel Tree Implementation

Advantages

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

Distributing Local Trees

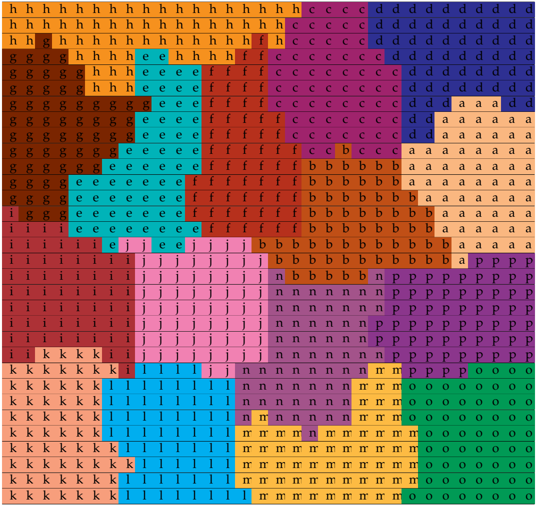
The interaction of locals trees is represented by a weighted graph.



This graph is partitioned, and trees assigned to processes.

Local Tree Distribution

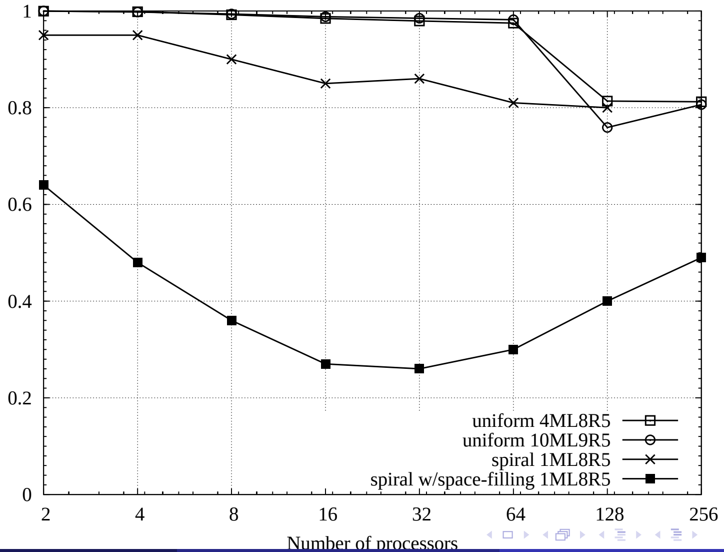
Here local trees are assigned to processes:



Parallel Data Movement

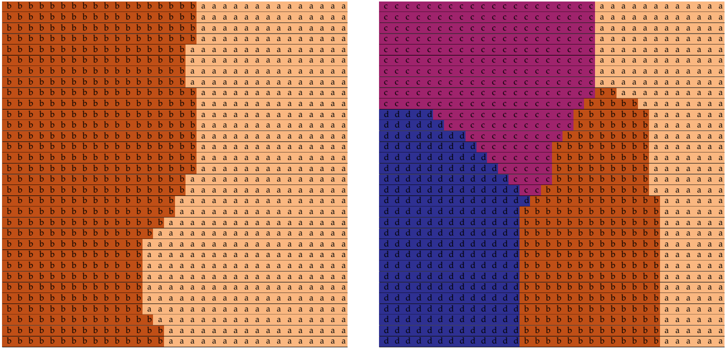
1. Complete neighbor section
2. Upward sweep
   1. Upward sweep on local trees
   2. Gather to root tree
   3. Upward sweep on root tree
3. Complete interaction list section
4. Downward sweep
   1. Downward sweep on root tree
   2. Scatter to local trees
   3. Downward sweep on local trees

PetFMM Load Balance



Local Tree Distribution

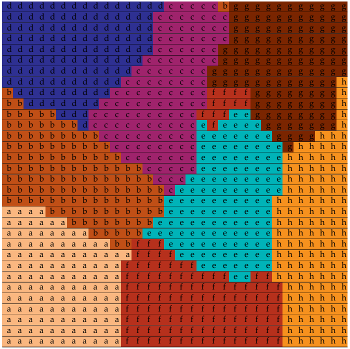
Here local trees are assigned to processes for a spiral distribution:



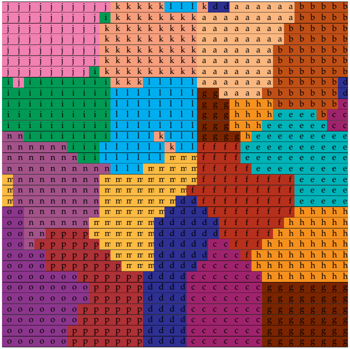
(a) 2 cores (b) 4 cores

Local Tree Distribution

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



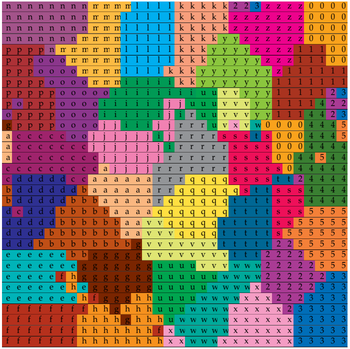
(c) 8 cores



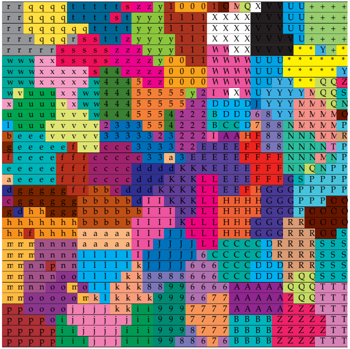
(d) 16 cores

Local Tree Distribution

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



(e) 32 cores



(f) 64 cores

Outline

1 Introduction

2 Short Introduction to FMM

3 Serial Implementation

4 Parallel FMM

5 Multicore FMM
- GPU Hardware
- PetFMM

Outline

5 Multicore FMM
- GPU Hardware
- PetFMM

GPU vs. CPU

A GPU looks like a big CPU with no virtual memory:

- Many more hardware threads encourage concurrency
- Makes bandwidth limitations even more acute
- *Shared memory* is really a user-managed cache
- *Texture memory* is also a specialized cache
- User also manages a very small code segment

GPU vs. CPU

Power usage can be very different:

Platform	TF	KW	GB/s	Price ($)	GF/$	GF/W
IBM BG/P	14	40.00	57.0*	1,800,000	0.008	0.35
IBM BlueGene	280	5000	???	350,000,000	0.0008	0.55
NVIDIA C1060	1	0.19	102.0	1,475	0.680	5.35
ATI 9250	1	0.12	63.5	840	1.220	8.33

Table: Comparison of Supercomputing Hardware.

Outline

5 Multicore FMM
- GPU Hardware
- PetFMM

GPU Performance

- In our C++ code on a CPU, M2L transforms take 85% of the time
  - This does vary depending on N
- New M2L design was implemented using PyCUDA
  - Port to C++ is underway
- We can now achieve 500 GF on the NVIDIA Tesla
  - Previous best performance we found was 100 GF
- We will release PetFMM-GPU in the new year

GPU Performance

- In our C++ code on a CPU, M2L transforms take 85% of the time
  - This does vary depending on N
- New M2L design was implemented using PyCUDA
  - Port to C++ is underway
- We can now achieve 500 GF on the NVIDIA Tesla
  - Previous best performance we found was 100 GF
- We will release PetFMM-GPU in the new year

GPU Performance

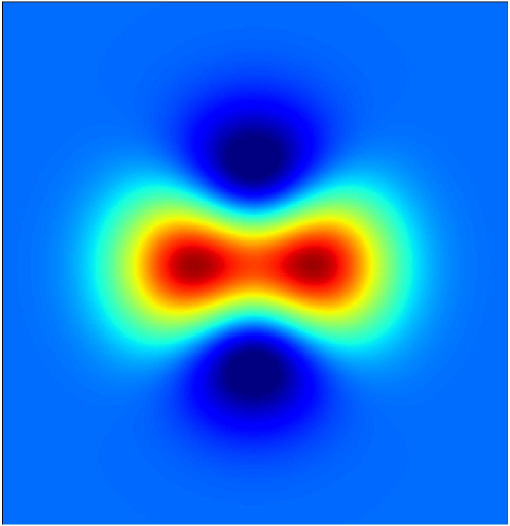
- In our C++ code on a CPU, M2L transforms take 85% of the time
  - This does vary depending on N
- New M2L design was implemented using PyCUDA
  - Port to C++ is underway
- We can now achieve 500 GF on the NVIDIA Tesla
  - Previous best performance we found was 100 GF
- We will release PetFMM-GPU in the new year

GPU Performance

- In our C++ code on a CPU, M2L transforms take 85% of the time
  - This does vary depending on N
- New M2L design was implemented using PyCUDA
  - Port to C++ is underway
- We can now achieve 500 GF on the NVIDIA Tesla
  - Previous best performance we found was 100 GF
- We will release PetFMM-GPU in the new year

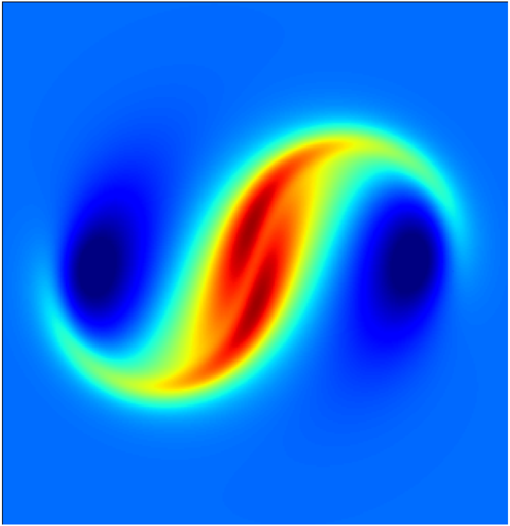
Tripolar Vortex

t = 000



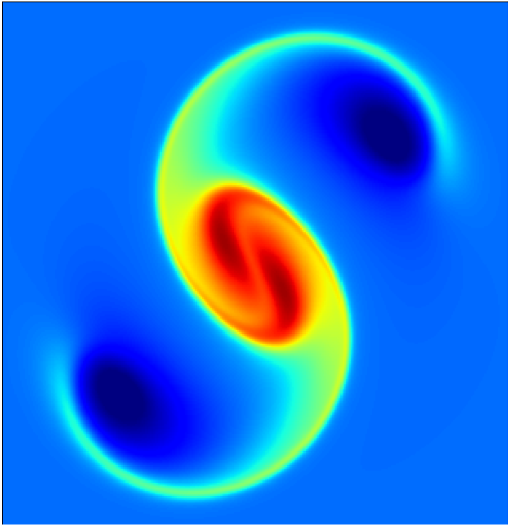
Tripolar Vortex

t = 100



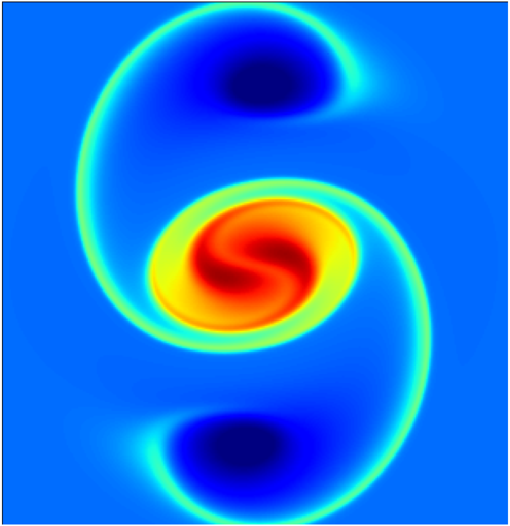
Tripolar Vortex

t = 200



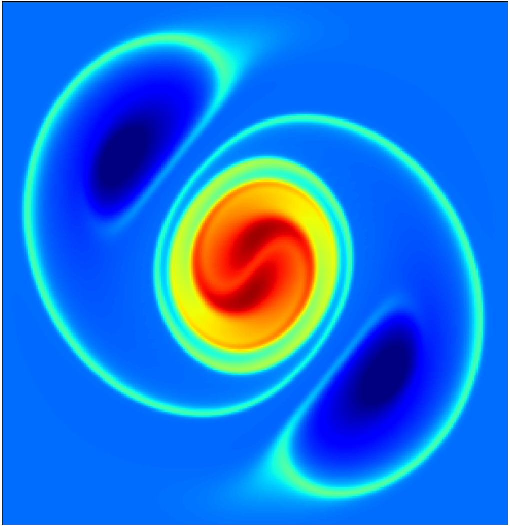
Tripolar Vortex

t = 300



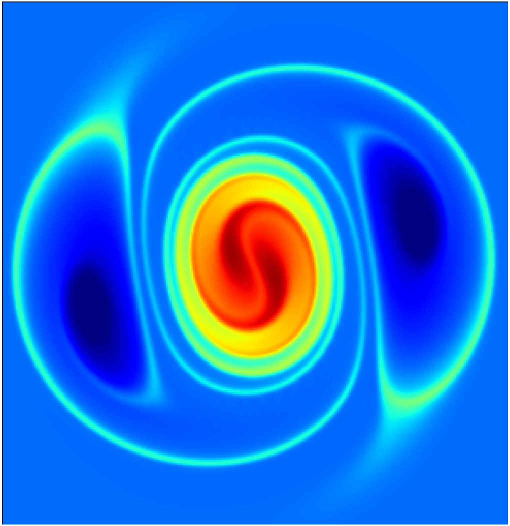
Tripolar Vortex

t = 400



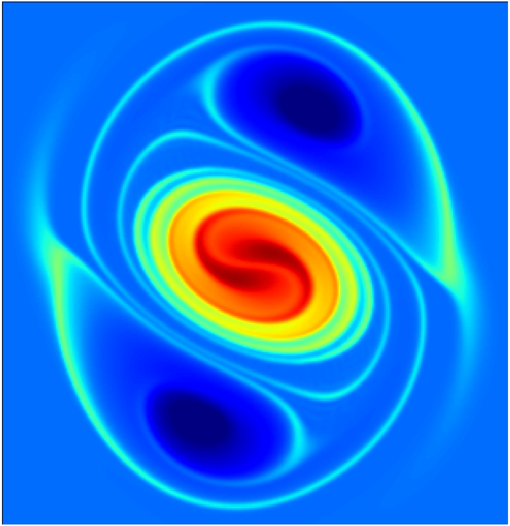
Tripolar Vortex

t = 500



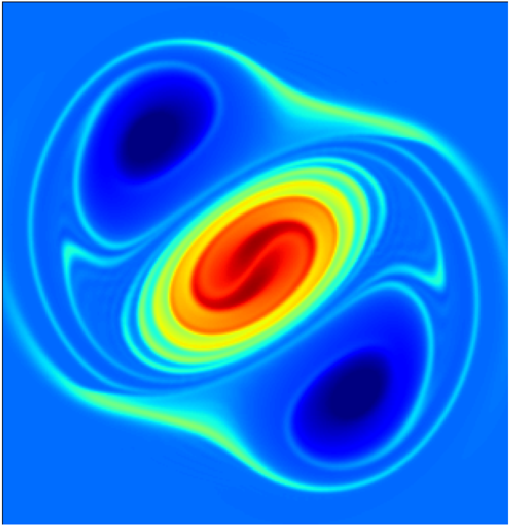
Tripolar Vortex

t = 600



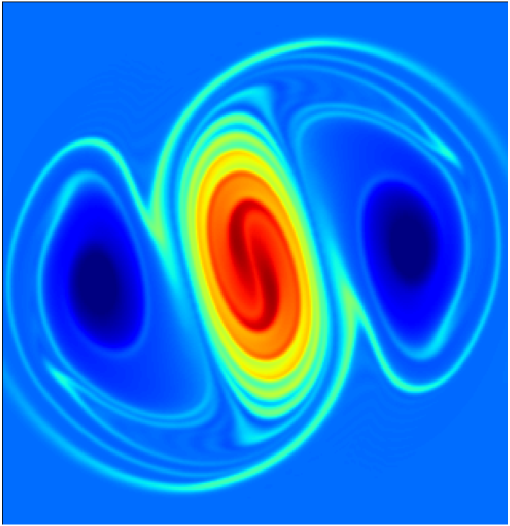
Tripolar Vortex

t = 700



Tripolar Vortex

t = 800



GPU Interaction

Since our parallelism is hierarchical

- Local (serial) tree interface is preserved
- GPU code can be reused locally without change
- Multiple GPUs per node can also be used

What's Important?

Interface improvements bring concrete benefits

- Facilitated code reuse
  - Serial code was largely reused
  - Test infrastructure completely reused
- Opportunites for performance improvement
  - Optimization using existing tools
  - Leverage GPU hardware
- Expansion of capabilities
  - Could now combine distributed and multicore implementations
  - Could replace local expansions with cheaper alternatives