

Tree-based methods on GPUs

Felipe Cruz¹ and Matthew Knepley^{2,3}

¹Department of Mathematics
University of Bristol

²Computation Institute
University of Chicago

³Department of Molecular Biology and Physiology
Rush University Medical Center

Department of Mathematics, LSU
Baton Rouge, LA Sept 25, 2009

Outline

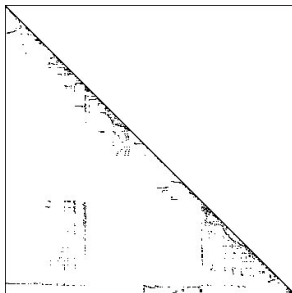
- 1 Introduction
- 2 Short Introduction to FMM
- 3 Serial Implementation
- 4 Multicore Interfaces
- 5 Multicore Implementation

Scientific Computing Challenge

How do we create
reusable
implementations which are also
efficient?

Structures are conserved,
but **tradeoffs** change.

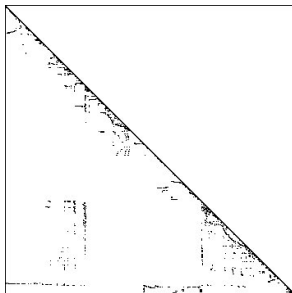
Structure vs. Tradeoffs



This is how **PETSc** works:

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

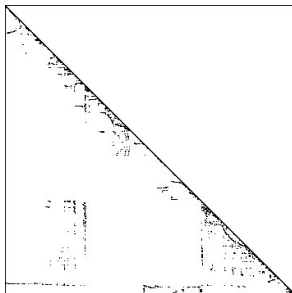
Structure vs. Tradeoffs



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

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

$$\{ \mathbf{b}, \mathbf{A}\mathbf{b}, \mathbf{A}(\mathbf{A}\mathbf{b}), \mathbf{A}(\mathbf{A}(\mathbf{A}\mathbf{b})), \dots \}$$

This is how **PETSc** works:

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

Structure vs. Tradeoffs

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

$$\{ \mathbf{b}, \mathbf{A}\mathbf{b}, \mathbf{A}(\mathbf{A}\mathbf{b}), \mathbf{A}(\mathbf{A}(\mathbf{A}\mathbf{b})), \dots \}$$

This is how **PETSc** works:

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

Structure vs. Tradeoffs

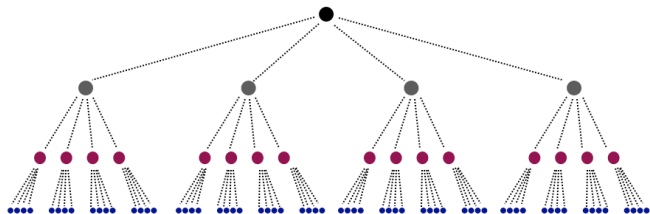
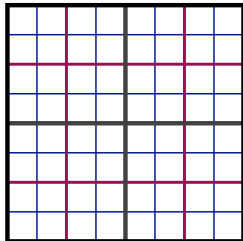
$$\mathbf{A} \mathbf{x} = \mathbf{b}$$

$$\{ \mathbf{b}, \mathbf{A}\mathbf{b}, \mathbf{A}(\mathbf{A}\mathbf{b}), \mathbf{A}(\mathbf{A}(\mathbf{A}\mathbf{b})), \dots \}$$

This is how **PETSc** works:

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

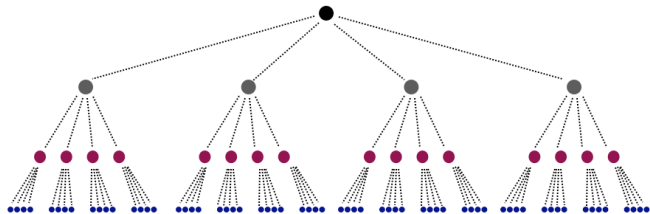
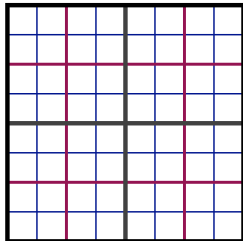
Structure vs. Tradeoffs



This is how treecodes work:

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

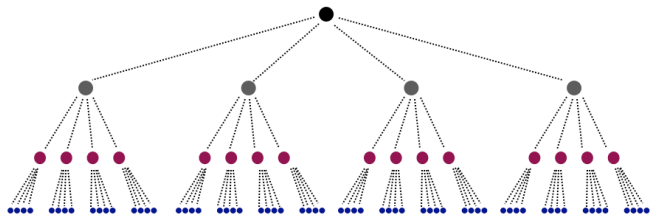
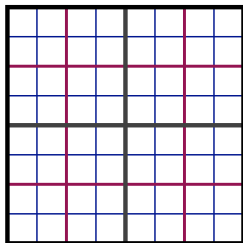
Structure vs. Tradeoffs



This is how treecodes work:

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

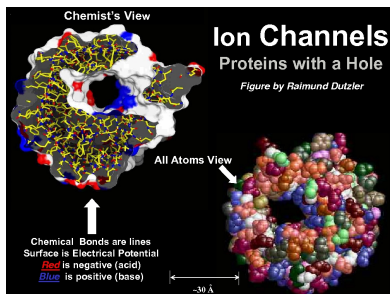
Structure vs. Tradeoffs



This is how treecodes work:

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

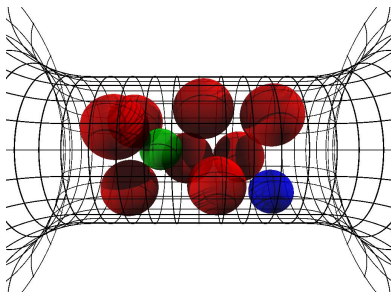
Structure vs. Tradeoffs



This is how biology works:

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

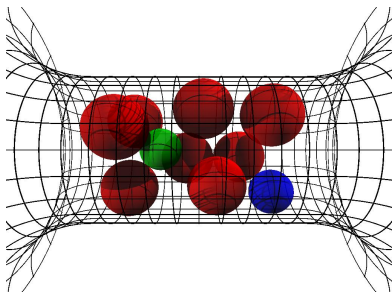
Structure vs. Tradeoffs



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 (FEniCS)

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

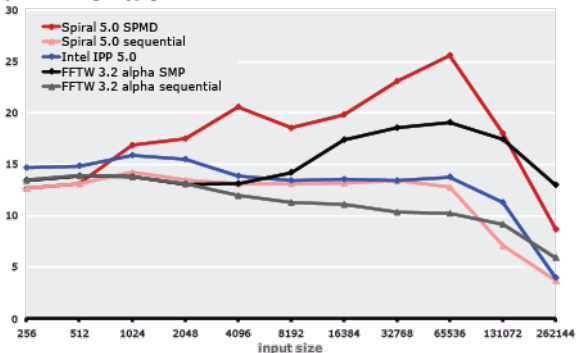
Treecodes:

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

Each level demands a strong abstraction layer

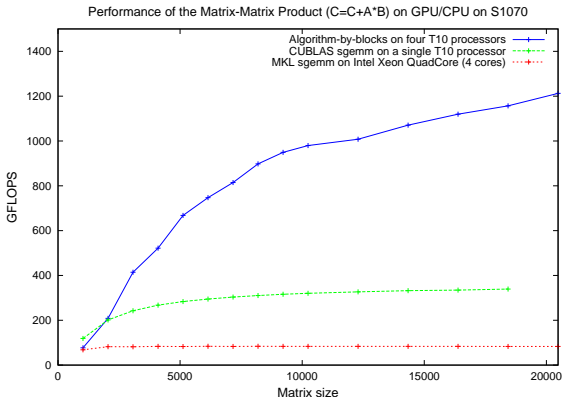
Spiral

DFT (single precision): on 3 GHz 2 x Core 2 Extreme
performance [Gflop/s]



- 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 Multicore Interfaces
- 5 Multicore Implementation

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_j)
 - Can be singular on the diagonal (Calderón-Zygmund operator)
- Discovered by Leslie Greengard and Vladimir Rokhlin in 1987
- Very similar to recent wavelet techniques

Fast Multipole Method

FMM accelerates the calculation of the function:

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

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

PetFMM

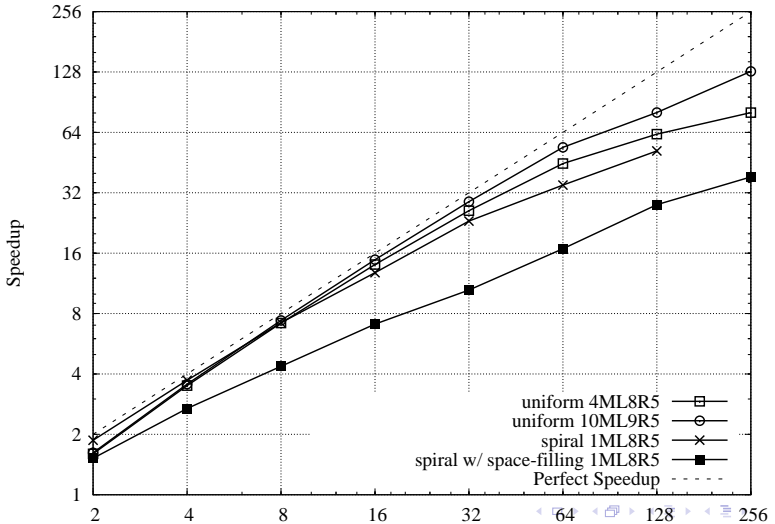
PetFMM is an freely available implementation of the
Fast **M**ultipole **M**ethod

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

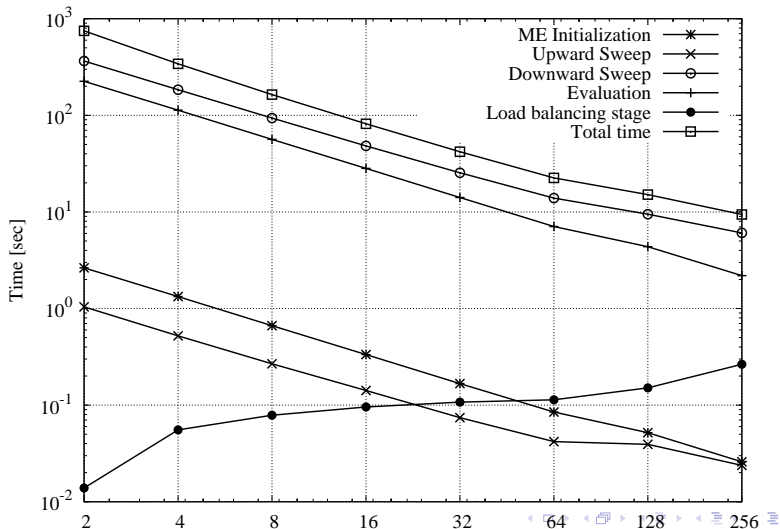
PetFMM CPU Performance

Strong Scaling



PetFMM CPU Performance

Strong Scaling



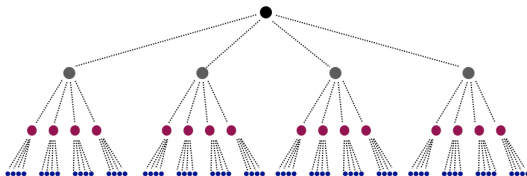
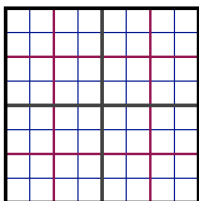
Outline

2 Short Introduction to FMM

- Spatial Decomposition
- Data Decomposition

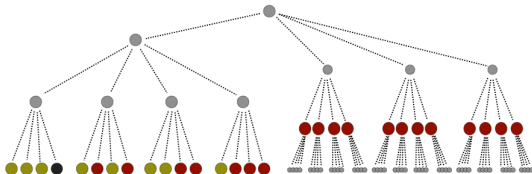
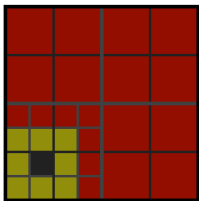
Spatial Decomposition

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



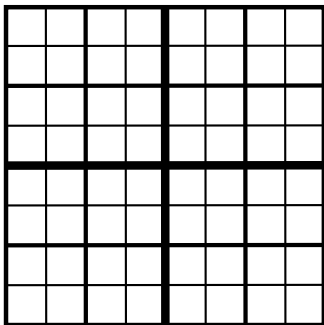
Spatial Decomposition

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



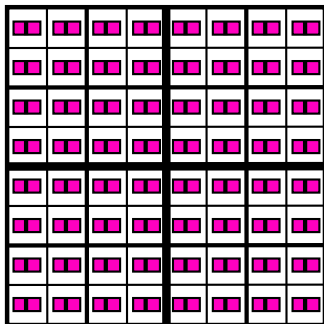
Neighbors are treated as *very near*.

FMM in Sieve



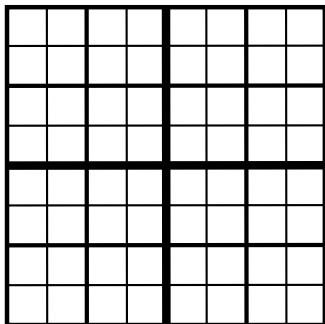
- The Quadtree is a Sieve
 - with optimized operations
- Multipoles are stored in Sections
- Two Overlaps are defined
 - Neighbors
 - Interaction List
- Completion moves data for
 - Neighbors
 - Interaction List

FMM in Sieve



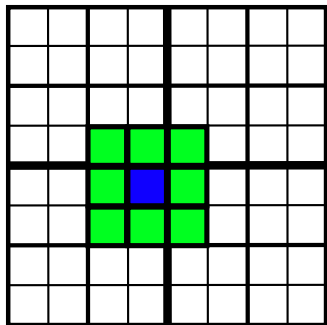
- The Quadtree is a Sieve
 - with optimized operations
- Multipoles are stored in Sections
- Two Overlaps are defined
 - Neighbors
 - Interaction List
- Completion moves data for
 - Neighbors
 - Interaction List

FMM in Sieve



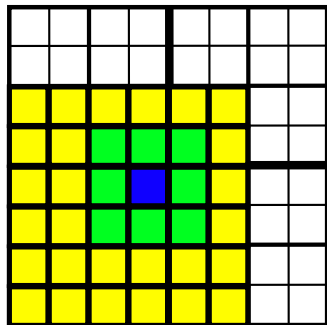
- The Quadtree is a Sieve
 - with optimized operations
- Multipoles are stored in Sections
- Two Overlaps are defined
 - Neighbors
 - Interaction List
- Completion moves data for
 - Neighbors
 - Interaction List

FMM in Sieve



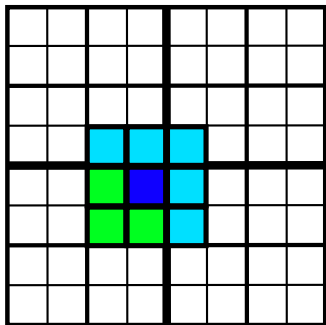
- The Quadtree is a Sieve
 - with optimized operations
- Multipoles are stored in Sections
- Two Overlaps are defined
 - Neighbors
 - Interaction List
- Completion moves data for
 - Neighbors
 - Interaction List

FMM in Sieve



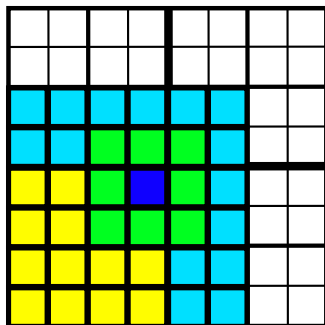
- The Quadtree is a Sieve
 - with optimized operations
- Multipoles are stored in Sections
- Two Overlaps are defined
 - Neighbors
 - Interaction List
- Completion moves data for
 - Neighbors
 - Interaction List

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

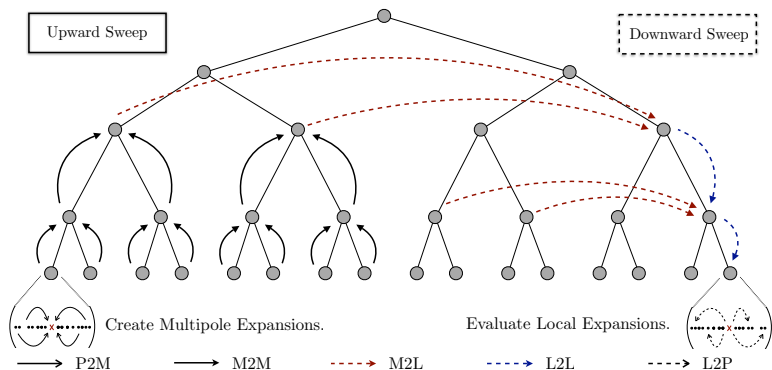
- 1 Introduction
- 2 Short Introduction to FMM
- 3 Serial Implementation**
 - Control Flow
 - Interface
- 4 Multicore Interfaces
- 5 Multicore Implementation

Outline

3 Serial Implementation

- Control Flow
- Interface

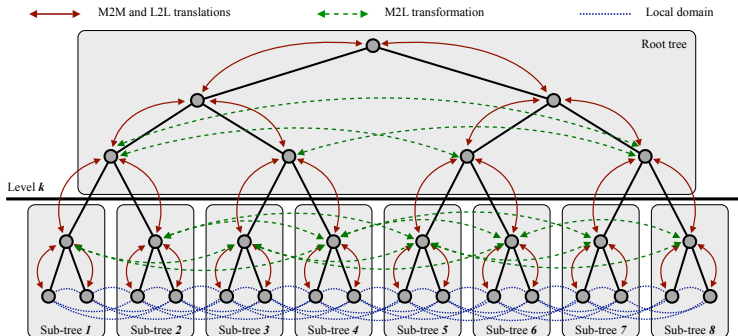
FMM Control Flow



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 Multicore Interfaces**
 - GPU Programming
 - FLASH
 - PetFMM
- 5 Multicore Implementation

Outline

- 4 Multicore Interfaces
 - GPU Programming
 - FLASH
 - 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.

STREAM Benchmark

Simple benchmark program measuring **sustainable** memory bandwidth

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

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

Table: Bandwidth limited machine performance

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

Analysis of Sparse Matvec (SpMV)

Assumptions

- No cache misses
- No waits on memory references

Notation

m Number of matrix rows

nz Number of nonzero matrix elements

V Number of vectors to multiply

We can look at bandwidth needed for peak performance

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

or achievable performance given a bandwidth BW

$$\frac{Vnz}{(8V + 2)m + 6nz} BW \text{ Mflop/s} \quad (3)$$

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

Improving Serial Performance

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

$$\frac{1}{(8 + 2)\frac{1}{7} + 6} \text{ bytes/flop} (1122.4 \text{ MB/s}) = \mathbf{151} \text{ MFlops/s}, \quad (4)$$

which is a dismal **8.8%** of peak.

Can improve performance by

- Blocking
- Multiple vectors

but operation issue limitations take over.

Improving Serial Performance

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

$$\frac{1}{(8 + 2)^{\frac{1}{7}} + 6} \text{ bytes/flop} (1122.4 \text{ MB/s}) = 151 \text{ MFlops/s}, \quad (4)$$

which is a dismal 8.8% of peak.

Better approaches:

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

GPU programming in General

- What design ideas are useful?
- How do we customize them for GPUs?
- Can we show an example?

Break Operations Into Small Chunks

Usually called **modularity**

- Also called *orthogonality* or *separation of concerns*
- Allows reduction of complexity
 - eXtreme programming
- Just concerned with functionality

Break Operations Into Small Chunks

GPU Differences

We now have to worry about **code size**!

- 16K total for NVIDIA 1060C board
 - Instructions can be a significant portion of memory usage
- Have to split operations which logically belong together
- Also allows aggregation of memory access
 - Computation can be regrouped
- Needs tools to manage many small tasks

Break Operations Into Small Chunks

Example

Reduction over a dataset

- For instance, computation of finite element integrals
- Break into *computation* and *aggregation* stages
- Model this by:
 - Maximum flop rate stage
 - Bandwidth limited stage

Break Operations Into Small Chunks

Example

Reduction over a dataset

- For instance, computation of Multipole-to-Local transform
- Break into *computation* and *aggregation* stages
- Model this by:
 - Maximum flop rate stage
 - Bandwidth limited stage

Reorder for Locality

Exploits “nearby” operations to aggregate computation

- Can be *temporal* or *spatial*
- Usually exploits a **cache**
- Difficult to predict/model on a modern processor

Reorder for Locality

GPU Differences

We have to manage our “cache” **explicitly**

- The NVIDIA 1060C shared memory is only 16K for 32 threads
- We must also manage “main memory” explicitly
 - Need to move data to/from GPU
- Must be aware of limited precision when reordering
- Can be readily modeled
- Need tools for automatic data movement (marshalling)

Reorder for Locality

Example

Data-Aware Work Queue

- A work queue manages many small tasks
 - Dependencies are tracked with a DAG
 - Queue should manage a single computational phase (supertask)
- Nodes also manage an input and output data segment
 - Specific classes can have known sizes
 - Can hold main memory locations for segments
- Framework manages marshalling:
 - Allocates contiguous data segments
 - Calculates segment offsets for tasks
 - Marshalls (moves) data
 - Passes offsets to supertask execution

Outline

- 4 Multicore Interfaces
 - GPU Programming
 - **FLASH**
 - PetFMM

FLASH Design

FLASH enables multicore computing through FLAME

- LA interface is identical to FLAME
- FLAME executes operations 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 operations 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 operations 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 operations 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

- 4 Multicore Interfaces
 - GPU Programming
 - FLASH
 - **PetFMM**

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:
 - $2(p+1) \text{ blockSize (Pascal)} + 2p \text{ blockSize (LE)} + 2p \text{ (ME)}$

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

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

CPU vs GPU

Sample run for 250,000 vortex particles in an 8 level tree

Section	Time(s)	
	PyCUDA	Laptop C++
Setup	0.55	0.00
InitExpansions	10.74	0.93
UpSweep	0.36	5.02
DownSweepEnqueue	0.09	—
GPUOverhead	2.97	—
DownSweepM2LTrns	2.08	363.21
DownSweepM2LRed	0.45	—
DownSweepL2L	0.36	4.11

Notice that once direct evaluation is moved to the GPU, Python can easily outperform C++.

Outline

- 1 Introduction
- 2 Short Introduction to FMM
- 3 Serial Implementation
- 4 Multicore Interfaces
- 5 Multicore Implementation**
 - Complexity Analysis
 - Redesign
 - MultiGPU

Outline

5 Multicore Implementation

- Complexity Analysis
- Redesign
- MultiGPU

Greengard & Gropp Analysis

For a shared memory machine,

$$T = a \frac{N}{P} + b \log_4 P + c \frac{N}{BP} + d \frac{NB}{P} + e(N, P) \quad (5)$$

- 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.

Outline

5 Multicore Implementation

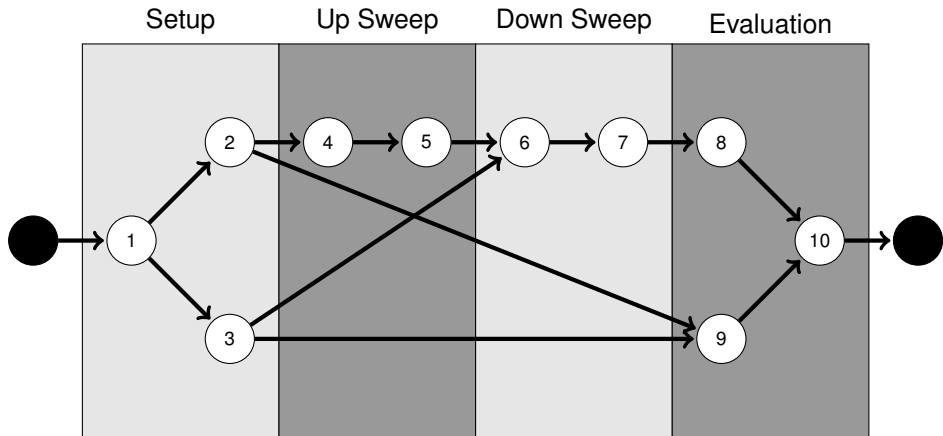
- Complexity Analysis
- Redesign
- MultiGPU

Question

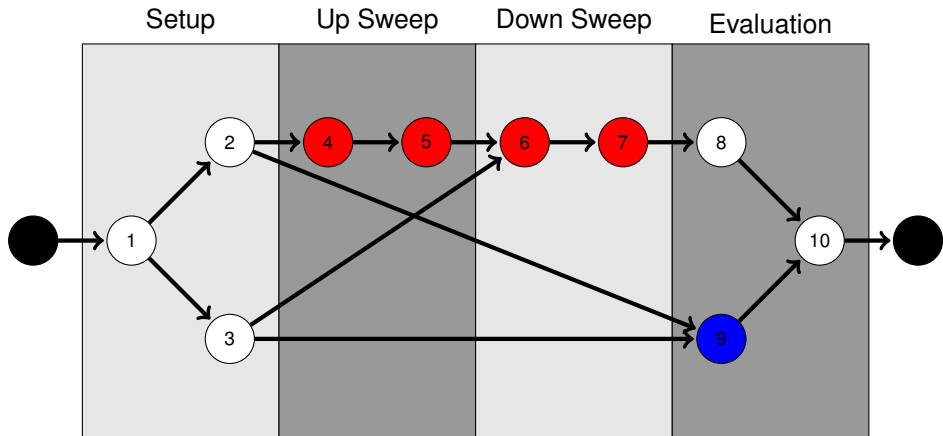
What is the optimal number of particles per cell?

- 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

PetFMM Stages



PetFMM Stages



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

$$B \geq \frac{b}{d} \frac{4^{L+1} p}{N} \quad (6)$$

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

$$M_{ij} = -1^i t^{-(i+j+1)} \binom{i+j}{j} \quad (7)$$

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

5 Multicore Implementation

- Complexity Analysis
- Redesign
- **MultiGPU**

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 dimension 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 $\mathcal{O}(\sqrt{n}(\log n)^{3/2})$ edgecut
 - 3D $\mathcal{O}(n^{2/3}(\log n)^{4/3})$ 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?

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 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

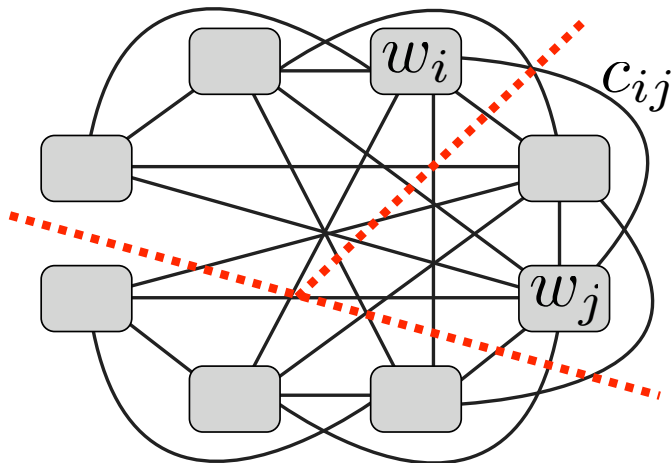
Parallel Tree Implementation

Advantages

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

Distributing Local Trees

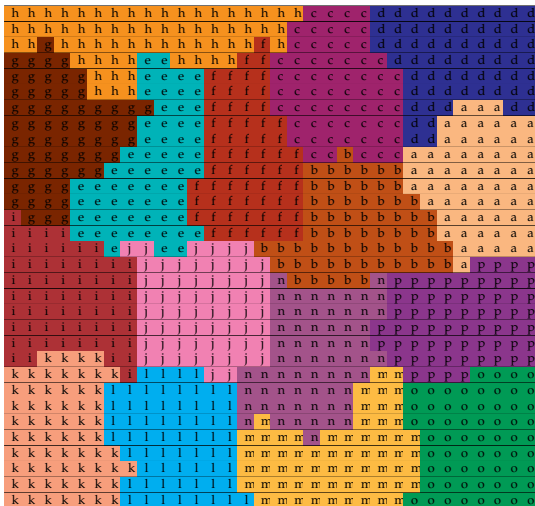
The interaction of local trees is represented by a weighted graph.



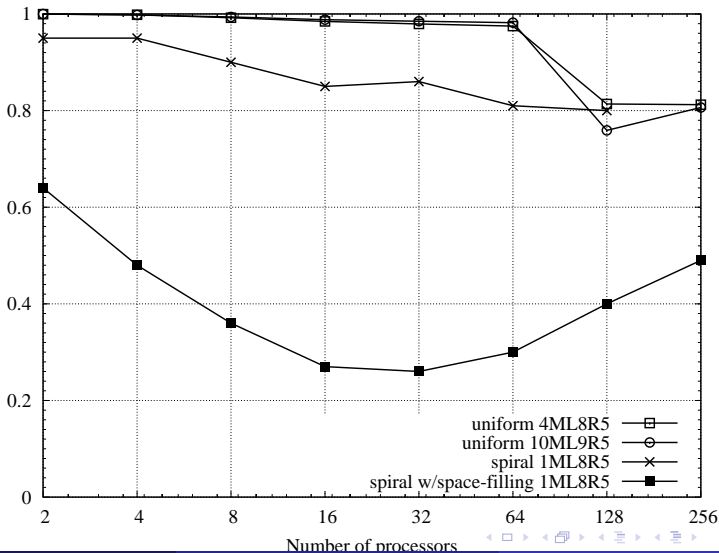
This graph is partitioned, and trees assigned to processes.

Local Tree Distribution

Here local trees are assigned to processes:

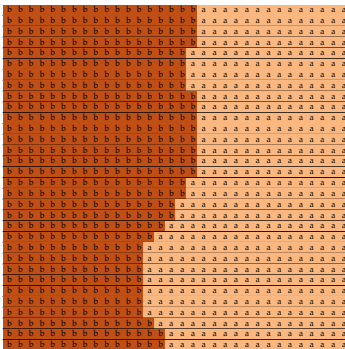


PetFMM Load Balance

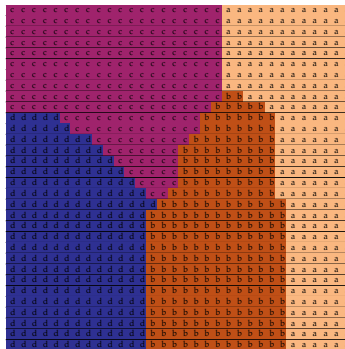


Local Tree Distribution

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



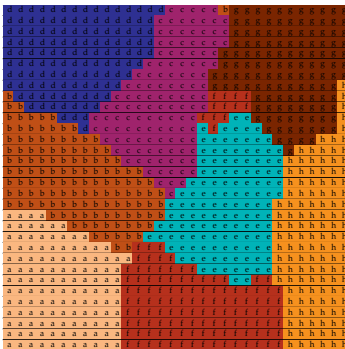
(a) 2 cores



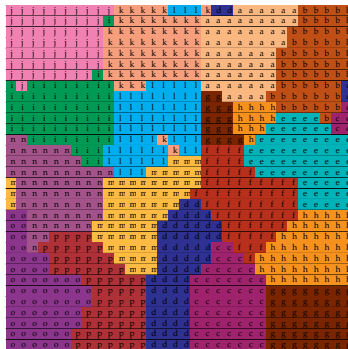
(b) 4 cores

Local Tree Distribution

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



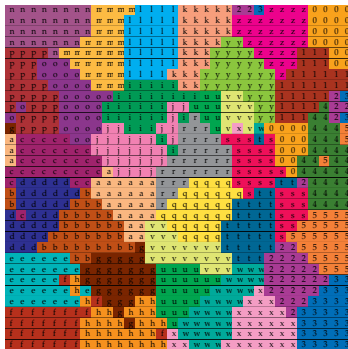
(c) 8 cores



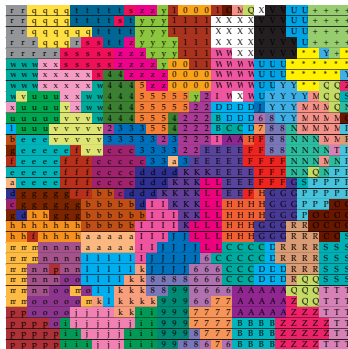
(d) 16 cores

Local Tree Distribution

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



(e) 32 cores



(f) 64 cores

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

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
- Opportunities 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