An Introduction to Fast Multipole Methods

Ramani Duraiswami
Institute for Advanced Computer Studies
University of Maryland, College Park
http://www.umiacs.umd.edu/~ramani

Joint work with Nail A. Gumerov

Fast Multipole Methods

- Computational simulation is becoming an accepted paradigm for scientific discovery.
  - Many simulations involve several million variables
- Most large problems boil down to solution of linear system or performing a matrix-vector product
- Regular product requires $O(N^2)$ time and $O(N^2)$ memory
- The FMM is a way to
  - accelerate the products of particular dense matrices with vectors
  - Do this using $O(N)$ memory
- FMM achieves product in $O(N)$ or $O(N \log N)$ time and memory
- Combined with iterative solution methods, can allow solution of problems hitherto unsolvable

Matrix vector product

$s_1 = m_{11} x_1 + m_{12} x_2 + \ldots + m_{1d} x_d$

$s_2 = m_{21} x_1 + m_{22} x_2 + \ldots + m_{2d} x_d$

$\ldots$

$s_n = m_{n1} x_1 + m_{n2} x_2 + \ldots + m_{nd} x_d$

- Matrix vector product is identical to a sum

  $s_i = \sum_{j=1}^{d} m_{ij} x_j$

- So algorithm for fast matrix vector products is also a fast summation algorithm

- d products and sums per line
- N lines
- Total Nd products and Nd sums to calculate N entries
- Memory needed is NM entries

Linear Systems

- Solve a system of equations

  $Mx=s$

- M is a $N \times N$ matrix, x is a N vector, s is a N vector
- Direct solution (Gauss elimination, LU Decomposition, SVD, …) all need $O(N^3)$ operations
- Iterative methods typically converge in k steps with each step needing a matrix vector multiply $O(N^2)$
  - ❑ if properly designed, $k << N$
- A fast matrix vector multiplication algorithm requiring $O(N \log N)$ operations will speed all these algorithms

Is this important?

- Argument:
  - Moore's law: Processor speed doubles every 18 months
  - If we wait long enough the computer will get fast enough and let my inefficient algorithm tackle the problem
- Is this true?
  - Yes for algorithms with same asymptotic complexity
  - No!! For algorithms with different asymptotic complexity
- For a million variables, we would need about 16 generations of Moore's law before a $O(N^2)$ algorithm is comparable with a $O(N)$ algorithm
- Similarly, clever problem formulation can also achieve large savings.

Memory complexity

- Sometimes we are not able to fit a problem in available memory
  - Don't care how long solution takes, just if we can solve it
- To store a $N \times N$ matrix we need N^2 locations
  - 1 GB RAM = 1024^3 =1,073,741,824 bytes
  - => largest N is 32,768
- "Out of core" algorithms copy partial results to disk, and keep only necessary part of the matrix in memory
  - Extremely slow
- FMM allows reduction of memory complexity as well
  - *Elements of the matrix required for the product can be generated as needed*
  - Can solve much larger problems (e.g., 10^7 variables on a PC)

The need for fast algorithms

- Grand challenge problems in large numbers of variables
- Simulation of physical systems
  - Electromagnetics of complex systems
  - Stellar clusters
  - Protein folding
  - Acoustics
  - Turbulence
- Learning theory
  - Kernel methods
  - Support Vector Machines
- Graphics and Vision
  - Light scattering …

- General problems in these areas can be posed in terms of millions (10^6) or billions (10^9) of variables
- Recall Avogadro's number ($6.022\ 141\ 99 \times 10^{23}$ molecules/mole
- Job of modeling is to find symmetries and representations that reduce the size of the problem
- Even after state of art modeling, problem size may be large

Dense and Sparse matrices

- Operation estimates are for **dense matrices**.
  - Majority of elements of the matrix are ***non-zero***
- However in many applications matrices are ***sparse***
- A sparse matrix (or vector, or array) is one in which most of the elements are zero.
  - If storage space is more important than access speed, it may be preferable to store a sparse matrix as a list of (index, value) pairs.
  - For a given sparsity structure it may be possible to define a fast matrix-vector product/linear system algorithm

- Can compute

$$\begin{bmatrix} a_1 & 0 & 0 & 0 & 0 \\ 0 & a_2 & 0 & 0 & 0 \\ 0 & 0 & a_3 & 0 & 0 \\ 0 & 0 & 0 & a_4 & 0 \\ 0 & 0 & 0 & 0 & a_5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} a_1x_1 \\ a_2x_2 \\ a_3x_3 \\ a_4x_4 \\ a_5x_5 \end{bmatrix}$$

In 5 operations instead of 25 operations

- Sparse matrices are not our concern here

Structured matrices

- Fast algorithms have been found for many dense matrices
- Typically the matrices have some *"structure"*
- Definition:
  - A dense matrix of order $N \times N$ is called structured if its entries depend on only $O(N)$ parameters.
- Most famous example – the fast Fourier transform

Fourier Matrices

A Fourier matrix of order n is defined as the following

$$F_n = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix},$$

where

$$\omega_n = e^{-\frac{2\pi i}{n}},$$

is an nth root of unity.

FFT presented by Cooley and Tukey in 1965, but invented several times, including by Gauss (1809) and Danielson & Lanczos (1948)

FFT and IFFT

The *discrete Fourier transform* of a vector x is the product $F_n x$.

The *inverse discrete Fourier transform* of a vector x is the product $F_n^* x$.

Both products can be done efficiently using the fast Fourier transform (FFT) and the inverse fast Fourier transform (IFFT) in $O(n \log n)$ time.

The FFT has revolutionized many applications by reducing the complexity by a factor of almost *n*

Can relate many other matrices to the Fourier Matrix

Circulant Matrices

$$C_n = C(x_1, ..., x_n) = \begin{bmatrix} x_1 & x_n & x_{n-1} & \cdots & x_2 \\ x_2 & x_1 & x_n & \cdots & x_3 \\ x_3 & x_2 & x_1 & \cdots & x_4 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_n & x_{n-1} & x_{n-2} & \cdots & x_1 \end{bmatrix}$$

Toeplitz Matrices

$$T_n = T(x_{-n+1}, \cdots, x_0, ..., x_{n-1}) = \begin{bmatrix} x_0 & x_1 & x_2 & \cdots & x_{n-1} \\ x_{-1} & x_0 & x_1 & \cdots & x_{n-2} \\ x_{-2} & x_{-1} & x_0 & \cdots & x_{n-3} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{-n+1} & x_{-n+2} & x_{-n+3} & \cdots & x_0 \end{bmatrix}$$

Hankel Matrices

$$H_n = H(x_{-n+1}, \cdots, x_0, \cdots, x_{n-1}) = \begin{bmatrix} x_{-n+1} & x_{-n+2} & x_{-n+3} & \cdots & x_0 \\ x_{-n+2} & x_{-n+3} & x_{-n+4} & \cdots & x_1 \\ x_{-n+3} & x_{-n+4} & x_{-n+5} & \cdots & x_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_0 & x_1 & x_2 & \cdots & x_{n-1} \end{bmatrix}$$

Vandermonde Matrices

$$V = V(x_0, x_1, ..., x_n) = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_0 & x_1 & \cdots & x_{n-1} \\ \cdots & \cdots & \cdots & \cdots \\ x_0^{n-1} & x_1^{n-1} & \cdots & x_{n-1}^{n-1} \end{bmatrix}$$

Structured Matrices

- (usually) these matrices can be diagonalized by the Fourier matrix
- Product of diagonal matrix and vector requires O(N) operations
- So complexity is the cost of FFT (O (N log N)) + product (O(N))
- Order notation
  - Only keep leading order term (asymptotically important)
  - So complexity of the above is O (N log N)
- Structured Matrix algorithms are "brittle"
  - FFT requires uniform sampling
  - Slight departure from uniformity breaks factorization

Fast Multipole Methods (FMM)

- Introduced by Rokhlin & Greengard in 1987
- Called one of the 10 most significant advances in computing of the 20th century
- Speeds up matrix-vector products (sums) of a particular type

$$s(x_j) = \sum_{i=1}^{N} \alpha_i \phi(x_j - x_i), \quad \{s_j\} = [\Phi_{ji}]\{\alpha_i\}.$$

- Above sum requires $O(MN)$ operations.
- For a given precision ε the FMM achieves the evaluation in $O(M+N)$ operations.
- Edelman: *"FMM is all about adding functions"*
  - *Talk on Tuesday, next week*

Is the FMM a structured matrix algorithm?

- FFT and other algorithms work on structured matrices
- What about FMM ?
- Speeds up matrix-vector products (sums) of a particular type

$$s(\mathbf{y}_j) = \sum_{i=1}^{N} a_i \phi(\mathbf{x}_i, \mathbf{y}_j)$$

$$\mathbf{s} = \mathbf{\Phi a} \qquad \{s_j\} = [\Phi_{ji}]\{\alpha_i\}$$

- Above sum also depends on $O(N)$ parameters $\{x_i\}$, $\{y_j\}$, ϕ
- FMM can be thought of as working on "loosely" structured matrices

- Can accelerate matrix vector products
  - Convert $O(N^2)$ to $O(N \log N)$
- However, can also accelerate linear system solution
  - Convert $O(N^3)$ to $O(kN \log N)$
  - For some iterative schemes can guarantee $k \leq N$
  - In general, goal of research in iterative methods is to reduce value of k
  - Well designed iterative methods can converge in very few steps
  - Active research area: design iterative methods for the FMM

A very simple algorithm

- Not FMM, but has some key ideas
- Consider

$$S(x_i)=\sum_{j=1}^{N} \alpha_j (x_i - y_j)^2 \qquad i=1, \ldots, M$$

- Naïve way to evaluate the sum will require *MN* operations
- Instead can write the sum as

$$S(x_i)=(\sum_{j=1}^{N} \alpha_j)x_i^2 + (\sum_{j=1}^{N} \alpha_j y_j^2) - 2x_i(\sum_{j=1}^{N} \alpha_j y_j)$$

  - Can evaluate each bracketed sum over *j* and evaluate an expression of the type

$$S(x_i)=\beta x_i^2 + \gamma - 2x_i\delta$$

  - Requires O(M+N) operations
- Key idea – use of analytical manipulation of series to achieve faster summation
- May not always be possible to simply factorize matrix entries

Approximate evaluation

- FMM introduces another key idea or "philosophy"
  - In scientific computing we almost never seek exact answers
  - At best, "exact" means to "machine precision"
- So instead of solving the problem we can solve a "nearby" problem that gives "almost" the same answer
  - If this "nearby" problem is much easier to solve, and we can bound the error analytically we are done.
- In the case of the FMM
  - Express functions in some appropriate functional space with a given basis
  - Manipulate series to achieve approximate evaluation
  - Use analytical expression to bound the error
- FFT is exact … FMM can be arbitrarily accurate

Approximation Algorithms

- Computer science approximation algorithms
  - Approximation algorithms are usually directed at reducing complexity of exponential algorithms by performing approximate computations
  - Here the goal is to reduce polynomial complexity to linear order
  - Connections between FMM and CS approximation algorithms are not much explored

Tree Codes

- Idea of approximately evaluating matrix vector products preceded FMM
- Tree codes (Barnes and Hut, 1986)
- Divides domain into regions and use approximate representations
- Key difference: lack error bounds, and automatic ways of adjusting representations
- Perceived to be easier to program

Complexity

- The most common complexities are
  - O(1) - not proportional to any variable number, i.e. a fixed/constant amount of time
  - O(N) - proportional to the size of N (this includes a loop to N and loops to constant multiples of N such as 0.5N, 2N, 2000N - no matter what that is, if you double N you expect (on average) the program to take twice as long)
  - O(N^2) - proportional to N squared (you double N, you expect it to take four times longer - usually two nested loops both dependent on N).
  - O(log N) - this is tricker to show - usually the result of binary splitting.
  - O(N log N) this is usually caused by doing log N splits but also doing N amount of work at each "layer" of splitting.

  - Exponential $O(a^N)$: grows faster than any power of N

Some FMM algorithms

- Molecular and stellar dynamics
  - Computation of force fields and dynamics
- Interpolation with Radial Basis Functions
- Solution of acoustical scattering problems
  - Helmholtz Equation
- Electromagnetic Wave scattering
  - Maxwell's equations
- Fluid Mechanics: Potential flow, vortex flow
  - Laplace/Poisson equations
- Fast nonuniform Fourier transform

Integral Equation

- FMM is often used in integral equations
- What is an integral equation?

$$\int k(x,y)u(x)dx + au(y) = f(y)$$

$$\int k(x,y)u(x)dx = f(y)$$

- Function *k(x,y)* is called the kernel
- Integral equations are typically solved by "quadrature"
  - Quadrature is the process of approximately evaluating an integral
- If we can write

$$\int k(x,y)u(x)dx = \sum_{j=1}^{N} k(x_j,y)u(x_j)w_j$$

FMM-able Matrices

$$\mathbf{v} = \mathbf{\Phi u},$$

$$\mathbf{\Phi} = \begin{pmatrix} \Phi(\mathbf{y}_1,\mathbf{x}_1) & \Phi(\mathbf{y}_1,\mathbf{x}_2) & \dots & \Phi(\mathbf{y}_1,\mathbf{x}_N) \\ \Phi(\mathbf{y}_2,\mathbf{x}_1) & \Phi(\mathbf{y}_2,\mathbf{x}_2) & \dots & \Phi(\mathbf{y}_2,\mathbf{x}_N) \\ \dots & \dots & \dots & \dots \\ \Phi(\mathbf{y}_M,\mathbf{x}_1) & \Phi(\mathbf{y}_M,\mathbf{x}_2) & \dots & \Phi(\mathbf{y}_M,\mathbf{x}_N) \end{pmatrix}.$$

$$\mathsf{X} = \{\mathbf{x}_1,\mathbf{x}_2,\dots,\mathbf{x}_N\}, \quad \mathbf{x}_i \in \mathsf{R}^d, \quad i = 1,\dots,N,$$
$$\mathsf{Y} = \{\mathbf{y}_1,\mathbf{y}_2,\dots,\mathbf{y}_M\}, \quad \mathbf{y}_j \in \mathsf{R}^d, \quad j = 1,\dots,M.$$

$$v_j = \sum_{i=1}^{N} u_i \Phi(\mathbf{y}_j,\mathbf{x}_i), \quad j = 1,\dots,M.$$

Factorization

Degenerate Kernel:

$$\Phi(\mathbf{y}_j, \mathbf{x}_i) = \sum_{m=0}^{p-1} A_m(\mathbf{x}_i) F_m(\mathbf{y}_j).$$

$$v_j = \sum_{i=1}^{N} u_i \Phi(\mathbf{y}_j, \mathbf{x}_i) = \sum_{i=1}^{N} u_i \sum_{m=0}^{p-1} A_m(\mathbf{x}_i) F_m(\mathbf{y}_j)$$
$$= \sum_{m=0}^{p-1} \left[\sum_{i=1}^{N} u_i A_m(\mathbf{x}_i) \right] F_m(\mathbf{y}_j) = \sum_{m=0}^{p-1} B_m F_m(\mathbf{y}_j).$$

$O(pN)$ operations:

$$B_m = \sum_{i=1}^{N} u_i A_m(\mathbf{x}_i), \quad m = 0, \ldots, p-1,$$

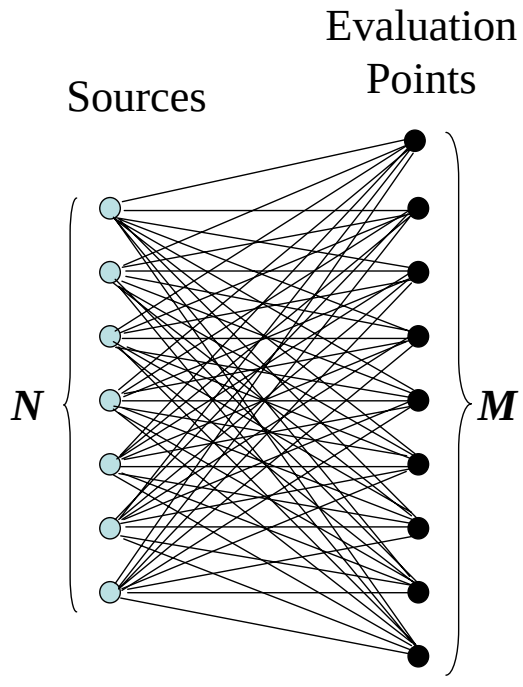
$O(pM)$ operations:

$$v_j = \sum_{m=0}^{p-1} B_m F_m(\mathbf{y}_j), \quad j = 1, \ldots, M.$$

Total Complexity: $O(p(N+M))$

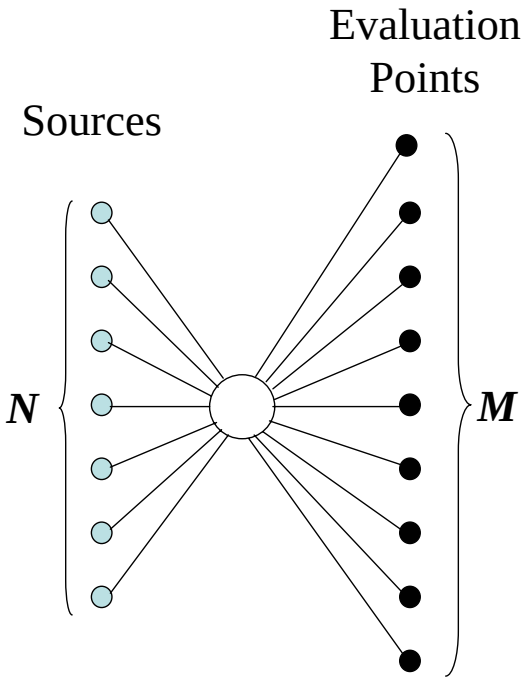
"Middleman" Algorithm

Standard algorithm



Total number of operations: ***O(NM)***

Middleman algorithm



Total number of operations: ***O(N+M)***

Factorization

Truncation Number

Non-Degenerate Kernel:

$$\Phi(\mathbf{y}_j,\mathbf{x}_i) = \sum_{m=0}^{p-1} A_m(\mathbf{x}_i)F_m(\mathbf{y}_j) + Error(p;\mathbf{x}_i,\mathbf{y}_j).$$

$$v_j = \sum_{i=1}^{N} u_i\Phi(\mathbf{y}_j,\mathbf{x}_i) = \sum_{i=1}^{N} u_i \sum_{m=0}^{p-1} A_m(\mathbf{x}_i)F_m(\mathbf{y}_j) + \sum_{i=1}^{N} u_i Error(p;\mathbf{x}_i,\mathbf{y}_j)$$
$$= \sum_{m=0}^{p-1} B_m F_m(\mathbf{y}_j) + Error_j(p,N), \quad j = 1,\ldots,M.$$

Error Bound: $|Error_j(p,N)| < N\max_i|u_i|\max_i\left|Error(p;\mathbf{x}_i,\mathbf{y}_j)\right|.$

Middleman Algorithm Applicability: $p \ll \min(M,N),$ $|Error_j(p,N)| < \epsilon.$

Factorization Problem:

- Usually there is no factorization available that provides a uniform approximation of the kernel in the entire computational domain.
- So we have to construct a patchwork-quilt of overlapping approximations, and manage this.

- Need representations of functions that allow this
- Need data structures for the management

Five Key Stones of FMM

- Representation and Factorization
- Error Bounds and Truncation
- Translation
- Space Partitioning
- Data Structures

Fast Multipole Methods

- Middleman (separation of variables)
  - No space partitioning
- Single Level Methods
  - Simple space partitioning (usually boxes)
- Multilevel FMM (MLFMM)
  - Multiple levels of space partitioning (usually hierarchical boxes)
- Adaptive MLFMM
  - Data dependent space partitioning

Examples of Matrices

- Green's functions of Laplace and Helmholtz equations

$$\Phi(\mathbf{y},\mathbf{x}) = \frac{1}{4\pi|\mathbf{y}-\mathbf{x}|},$$

$$\Phi(\mathbf{y},\mathbf{x}) = \frac{\exp\{ik|\mathbf{y}-\mathbf{x}|\}}{4\pi|\mathbf{y}-\mathbf{x}|}.$$

- Potential velocity field of a source located at $\mathbf{x}_i$

$$\mathbf{\Phi}(\mathbf{y},\mathbf{x}_i) = \mathbf{V}(\mathbf{y},\mathbf{x}_i) = \frac{1}{4\pi}\nabla_{\mathbf{y}}\frac{1}{|\mathbf{y}-\mathbf{x}_i|}.$$

- Normal derivative on the surface

$$\Phi(\mathbf{y},\mathbf{x}) = \frac{\partial}{\partial n(\mathbf{x})}\frac{1}{4\pi|\mathbf{y}-\mathbf{x}|} = \mathbf{n}(\mathbf{x})\bullet\nabla_{\mathbf{x}}\frac{1}{4\pi|\mathbf{y}-\mathbf{x}|}.$$

- Vorticity (vortex element is located at $\mathbf{x}_i$)

$$\Phi(\mathbf{y},\mathbf{x}_i) = \nabla_{\mathbf{y}}\times\mathbf{V}(\mathbf{y},\mathbf{x}_i).$$

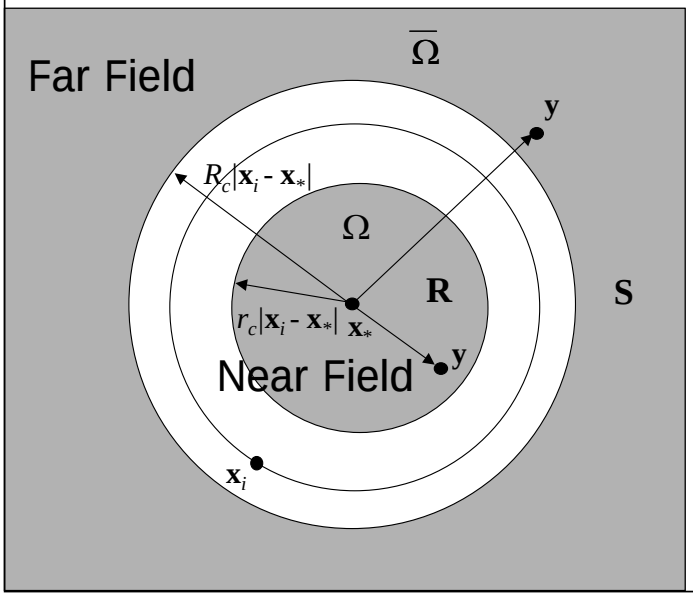
Iterative Methods

- To solve linear systems of equations;
- Simple iteration methods;
- Conjugate gradient or similar methods;
- We use Krylov subspace methods:
  - ❑Parameters of the method;
  - ❑Preconditioners;
  - ❑Research is ongoing.
- Efficiency critically depends on efficiency of the matrix-vector multiplication.

Far and Near Field Expansions

Far Field: $$\Phi(\mathbf{y}_j,\mathbf{x}_i) = \sum_{m=0}^{p-1} C_m(\mathbf{x}_i,\mathbf{x}_*) S_m(\mathbf{y}_j - \mathbf{x}_*) + Error.$$

Near Field: $$\Phi(\mathbf{y}_j,\mathbf{x}_i) = \sum_{m=0}^{p-1} D_m(\mathbf{x}_i,\mathbf{x}_*) R_m(\mathbf{y}_j - \mathbf{x}_*) + Error.$$



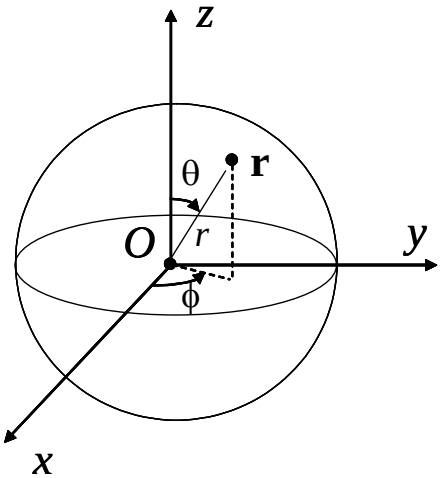
S: "Singular"
(also "Multipole",
"Outer"
"Far Field"),

R: "Regular"
(also "Local",
"Inner"
"Near Field")

Example of Multipole and Local expansions (3D Laplace)

$$\Phi(\mathbf{y}_j,\mathbf{x}_i) = \sum_{n=0}^{p-1}\sum_{m=-n}^{n} C_n^m S_n^m(\mathbf{y}_j - \mathbf{x}_*) + Error(p),$$

$$\Phi(\mathbf{y}_j,\mathbf{x}_i) = \sum_{n=0}^{p-1}\sum_{m=-n}^{n} D_n^m R_n^m(\mathbf{y}_j - \mathbf{x}_*) + Error(p),$$



$$S_n^m(\mathbf{r}) = \frac{(-1)^n i^{|m|}}{\alpha_n^m}\sqrt{\frac{4\pi}{2n+1}}\frac{1}{r^{n+1}}Y_n^m(\theta,\varphi),$$

$$R_n^m(\mathbf{r}) = i^{-|m|}\alpha_n^m\sqrt{\frac{4\pi}{2n+1}}\,r^n Y_n^m(\theta,\varphi),$$

$$\alpha_n^m = \alpha_n^{-m} = \frac{(-1)^n}{\sqrt{(n-m)!(n+m)!}}.$$

Spherical Coordinates:

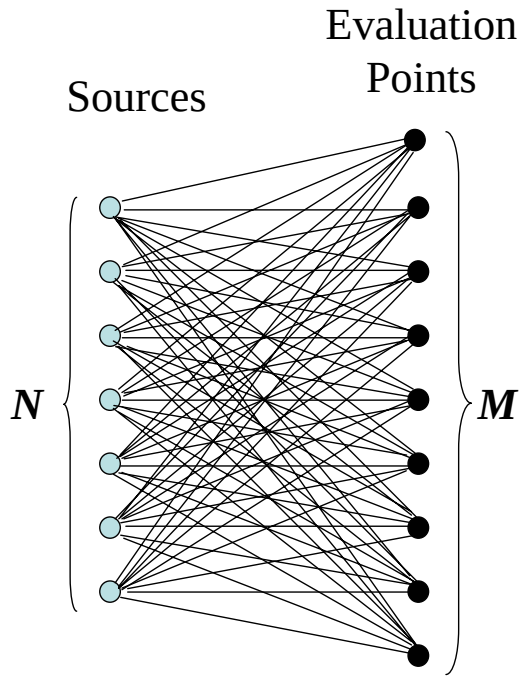
$$\mathbf{r} = r(\sin\theta\cos\varphi, \sin\theta\sin\varphi, \cos\theta)$$

Spherical Harmonics:

$$Y_n^m(\theta,\varphi) = (-1)^m\sqrt{\frac{2n+1}{4\pi}\frac{(n-|m|)!}{(n+|m|)!}}\,P_n^{|m|}(\cos\theta)e^{im\varphi}$$

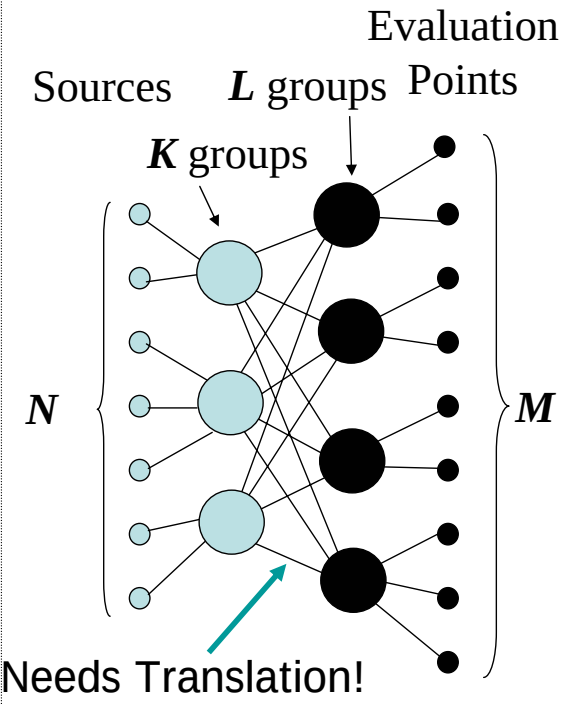
Idea of a Single Level FMM

Standard algorithm



Total number of operations: ***O(NM)***

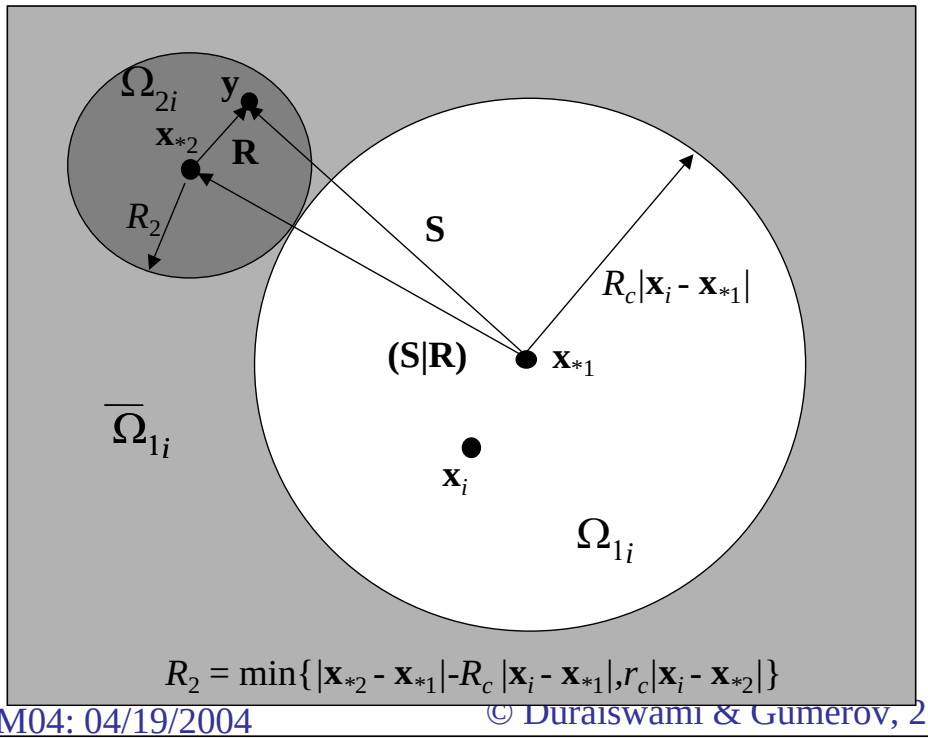
SLFMM



Total number of operations: ***O(N+M+KL)***

Multipole-to-Local S|R-translation

Also “Far-to-Local”, “Outer-to-Inner”, “Multipole-to-Local”



$$R_2 = \min\{|\mathbf{x}_{*2} - \mathbf{x}_{*1}| - R_c|\mathbf{x}_i - \mathbf{x}_{*1}|, r_c|\mathbf{x}_i - \mathbf{x}_{*2}|\}$$

S|R-translation Operator

$$\Phi(\mathbf{y}) = \sum_{m=0}^{p-1} C_m S_m(\mathbf{y} - \mathbf{x}_{*1}) + Error.$$

$$\Phi(\mathbf{y}) = \sum_{m=0}^{p-1} D_m R_m(\mathbf{y} - \mathbf{x}_{*2}) + Error.$$

S|R-Translation Coefficients

$$S_n(\mathbf{y} - \mathbf{x}_{*1}) = \sum_{m=0}^{p-1} (S|R)_{mn}(\mathbf{x}_{*2} - \mathbf{x}_{*1}) R_m\left(\mathbf{y}_j - \mathbf{x}_{*2}\right) + Error.$$

S|R-Translation Matrix

$$D_m(\mathbf{y} - \mathbf{x}_{*1}) = \sum_{m=0}^{p-1} (S|R)_{mn}(\mathbf{x}_{*2} - \mathbf{x}_{*1}) C_n\left(\mathbf{y}_j - \mathbf{x}_{*2}\right) + Error.$$

S|R-translation Operators for 3D Laplace and Helmholtz equations

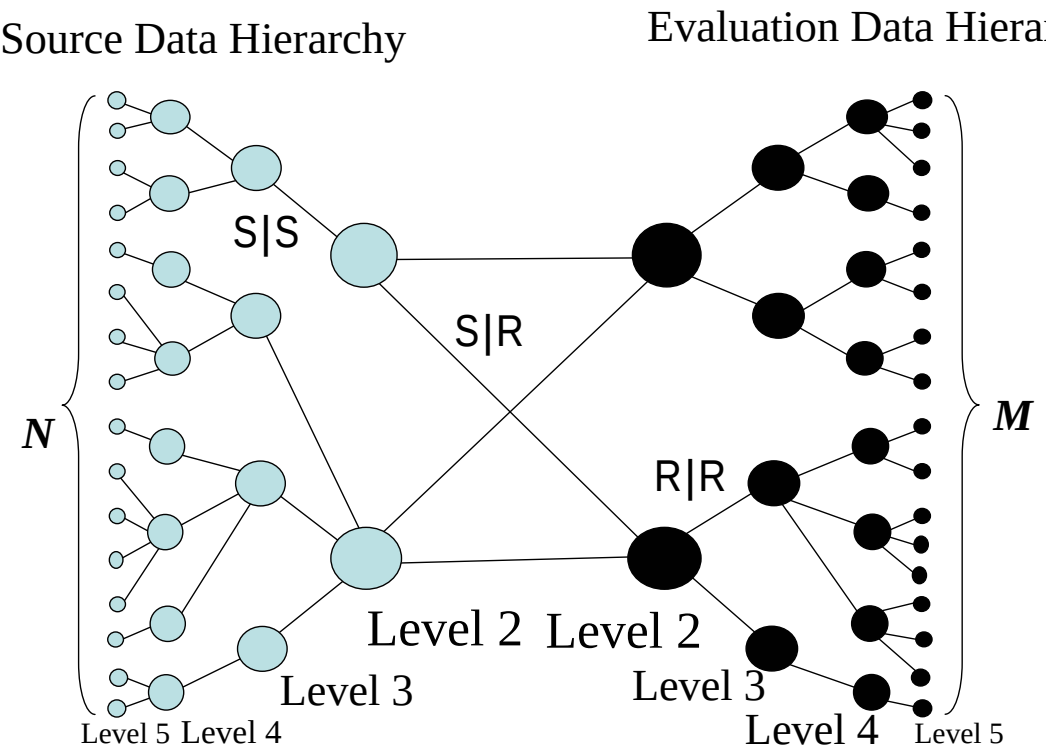
$$\Phi(\mathbf{y}) = \sum_{n=0}^{p-1} \sum_{m=-n}^{n} C_n^m S_n^m(\mathbf{y} - \mathbf{x}_{*1}) + Error.$$

$$\Phi(\mathbf{y}) = \sum_{n=0}^{p-1} \sum_{m=-n}^{n} D_n^m R_n^m(\mathbf{y} - \mathbf{x}_{*2}) + Error.$$

$$S_n^m(\mathbf{y} - \mathbf{x}_{*1}) = \sum_{n'=0}^{p-1} \sum_{m'=-n'}^{n'} (S|R)_{n'n}^{m'm}(\mathbf{x}_{*2} - \mathbf{x}_{*1}) R_{n'}^{m'}\left(\mathbf{y}_j - \mathbf{x}_{*2}\right) + Error.$$

$$D_n^m(\mathbf{y} - \mathbf{x}_{*1}) = \sum_{n'=0}^{p-1} \sum_{m'=-n'}^{n'} (S|R)_{nn'}^{mm'}(\mathbf{x}_{*2} - \mathbf{x}_{*1}) C_{n'}^{m'}\left(\mathbf{y}_j - \mathbf{x}_{*2}\right) + Error.$$

Idea of Multilevel FMM



Complexity of Translation

- For 3D Laplace and Helmholtz series have p^2 terms;
- Translation matrices have p^4 elements;
- Translation performed by direct matrix-vector multiplication has complexity $O(p^4)$;
- Can be reduced to $O(p^3)$;
- Can be reduced to $O(p^2 \log^2 p)$;
- Can be reduced to $O(p^2)$ (?).

Week 2: Representations

- **Gregory Beylkin** (University of Colorado) "Separated Representations and Fast Adaptive Algorithms in Multiple Dimensions"
- **Alan Edelman** (MIT) "Fast Multipole: It's All About Adding Functions in Finite Precision"
- **Vladimir Rokhlin** (Yale University) "Fast Multipole Methods in Oscillatory Environments: Overview and Current State of Implementation"
- **Ramani Duraiswami** (University of Maryland) "An Improved Fast Gauss Transform and Applications"
- **Eric Michielssen** (University of Illinois at Urbana-Champaign) "Plane Wave Time Domain Accelerated Integral Equation Solvers"

Week 2: Data Structures

- **David Mount** (University of Maryland) "Data Structures for Approximate Proximity and Range Searching"
- **Alexander Gray** (Carnegie Mellon University) "New Lightweight N-body Algorithms"
- **Ramani Duraiswami** (University of Maryland) "An Improved Fast Gauss Transform and Applications"

Week 2: Applications

- **Nail Gumerov** (University of Maryland) "Computation of 3D Scattering from Clusters of Spheres using the Fast Multipole Method"
- **Weng Chew** (University of Illinois at Urbana-Champaign) "Review of Some Fast Algorithms for Electromagnetic Scattering"
- **Leslie Greengard** (Courant Institute, NYU) "FMM Libraries for Computational Electromagnetics"
- **Qing Liu** (Duke University) "NUFFT, Discontinuous Fast Fourier Transform, and Some Applications"
- **Eric Michielssen** (University of Illinois at Urbana-Champaign) "Plane Wave Time Domain Accelerated Integral Equation Solvers"
- **Gregory Rodin** (University of Texas, Austin) "Periodic Conduction Problems: Fast Multipole Method and Convergence of Integral Equations and Lattice Sums"
- **Stephen Wandzura** (Hughes Research Laboratories) "Fast Methods for Fast Computers"
- **Toru Takahashi** (Institue of Physical and Chemical Research (RIKEN), Japan) "Fast Computing of Boundary Integral Equation Method by a Special-purpose Computer"
- **Ramani Duraiswami** (University of Maryland) "An Improved Fast Gauss Transform and Applications"

Tree Codes:

- **Atsushi Kawai** (Saitama Institute of Technology) "Fast Algorithms on GRAPE Special-Purpose Computers"
- **Walter Dehnen** (University of Leicester) "falcON: A Cartesian FMM for the Low-Accuracy Regime"
- **Robert Krasny** (University of Michigan) "A Treecode Algorithm for Regularized Particle Interactions“
- **Derek Richardson** (University of Maryland) "pkdgrav: A Parallel k-D Tree Gravity Solver for N-Body Problems"