

Fast Multipole Methods: Fundamentals & Applications

Ramani Duraiswami
Nail A. Gumerov

Grade distribution

- Homework 35 % of your grade
- Projects 25%
- Final 25%
- Mid-term 15 %

Problem Size

- As problem size grows, in general fidelity of representation grows
- On the other hand
 - cost of solving the problem
 - Memory required to store the discrete representation also grow
- Way the cost creases with size is the “memory” or “run time” complexity of an algorithm
- Need a language to talk about fast and low memory algorithms
- Asymptotic complexity

Asymptotic Equivalence

- Let n the problem size
- Two algorithms with costs $f(n)$ and $g(n)$ are said to be equivalent:

$$f(n) \sim g(n) \quad \lim_{n \rightarrow \infty} \left(\frac{f(n)}{g(n)} \right) = 1$$

Little Oh

• *Asymptotically smaller:*

$$\bullet f(n) = o(g(n))$$

$$\lim_{n \rightarrow \infty} \left(\frac{f(n)}{g(n)} \right) = 0$$

Big Oh

• *Asymptotic Order of Growth:*

$$\bullet f(n) = O(g(n))$$

$$\limsup_{n \rightarrow \infty} \left(\frac{f(n)}{g(n)} \right) < \infty$$

The Oh's

If $f = o(g)$ or $f \sim g$ then $f = O(g)$

$$\lim = 0 \quad \lim = 1 \quad \lim < \infty$$

The Oh's

If $f = o(g)$, then $g \neq O(f)$

$$\lim \frac{f}{g} = 0 \quad \lim \frac{g}{f} = \infty$$

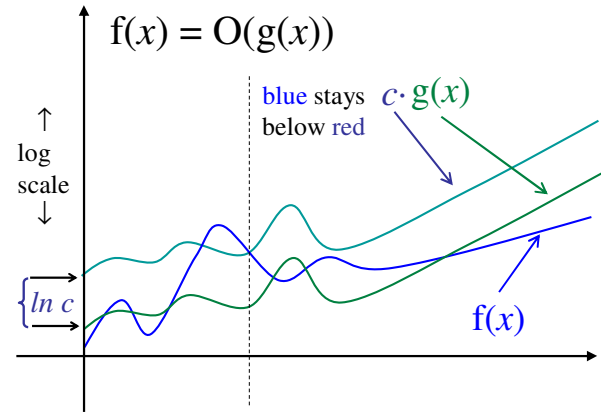
Big Oh

•Equivalently,

$$\bullet f(n) = O(g(n))$$

$$\exists c, n_0 \geq 0 \forall n \geq n_0 |f(n)| \leq c \cdot g(n)$$

Big Oh



Complexity

- we will specify the function $g(N)$
- 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 trickier 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.

Theta

Same Order of Growth:

$$\bullet f(n) = \Theta(g(n))$$

$$f(n) = O(g(n)) \text{ and } g(n) = O(f(n))$$

Log complexity

- If you half data at each stage then number of stages until you have a single item is given (roughly) by $\log_2 N$. \Rightarrow binary search takes $\log_2 N$ time to find an item.
- All logs grow a constant amount apart (homework)
 - So we normally just say $\log N$ not $\log_2 N$.
- $\log N$ grows very slowly

Vectors and Matrices

d dimensional column vector \mathbf{x} and its transpose

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} \quad \text{and} \quad \mathbf{x}^t = (x_1 \ x_2 \ \dots \ x_d)$$

• $n \times d$ dimensional matrix \mathbf{M} and its transpose \mathbf{M}^t

$$\mathbf{M} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1d} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & m_{n3} & \dots & m_{nd} \end{pmatrix} \quad \text{and}$$

$$\mathbf{M}^t = \begin{pmatrix} m_{11} & m_{12} & \dots & m_{n1} \\ m_{12} & m_{22} & \dots & m_{n2} \\ m_{13} & m_{23} & \dots & m_{n3} \\ \vdots & \vdots & \ddots & \vdots \\ m_{1d} & m_{2d} & \dots & m_{nd} \end{pmatrix}.$$

Matrix vector product

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

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

...

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

- 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
- If $d \sim N$ then matrix multiplication cost is $O(N^2)$
- Storage of the matrix entries is also $O(N^2)$

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.

- Can accelerate matrix vector products
 - Convert $O(N^2)$ to $O(N \log N)$
- However, can also accelerate linear system solution
 - Use iterative methods that take k steps and use a matrix vector product at each step
 - Convert $O(N^3)$ to $O(kN \log N)$

A very simple algorithm

- Not FMM, but has some key ideas
- Consider

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

- 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
- Matrix is structured ... determined by $N+M$ quantities

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.

Reduction of Complexity

Straightforward (nested loops):

```

for j = 1, ..., M
  v_j = 0;
  for i = 1, ..., N
    v_j = v_j + \Phi(y_j, x_i) u_i;
  end;
end;

```

Complexity: $O(MN)$

Factorize matrix entries

$$\Phi(y_j, x_i) = \sum_{m=0}^{p-1} a_m(\mathbf{x}_i - \mathbf{x}_*) f_m(y_j - \mathbf{x}_*)$$

If $p \ll \min(M, N)$ then complexity reduces!

Factorized:

```

for m = 0, ..., p-1
  c_m = 0;
  for i = 1, ..., N
    c_m = c_m + a_m(x_i - x_*) u_i;
  end;
end;

```

```

for j = 1, ..., M
  v_j = 0;
  for m = 0, ..., p-1
    v_j = v_j + c_m f_m(y_j - x_*);
  end;
end;

```

Complexity: $O(pN+pM)$

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
 - Manipulate series to achieve approximate evaluation
 - Use analytical expression to bound the error
- FFT is exact ... FMM can be arbitrarily accurate

Approximate evaluation

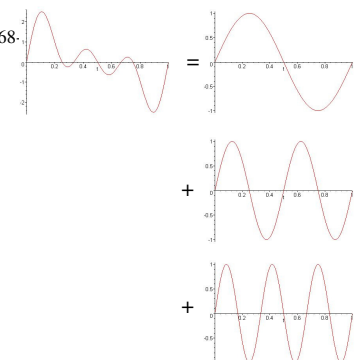
- FMM introduces another key idea or “philosophy”
 - In scientific computing we almost never seek exact answers
 - At best, “exact” means to “machine precision”
- So instead of solving the problem we can solve a “nearby” problem that gives “almost” the same answer
- If this “nearby” problem is much easier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

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
- FMM matrices are also structured
 - Depend on the $N+M$ points

Fourier Analysis

- Def.: mathematical techniques for breaking up a signal into its components (sinusoids)
- Jean Baptiste Joseph Fourier (1768-1830)
- can represent any continuous periodic signal as a sum of sinusoidal waves



Notation

For f , periodic with period p

Fourier transform $f(t) \rightarrow F[k]$

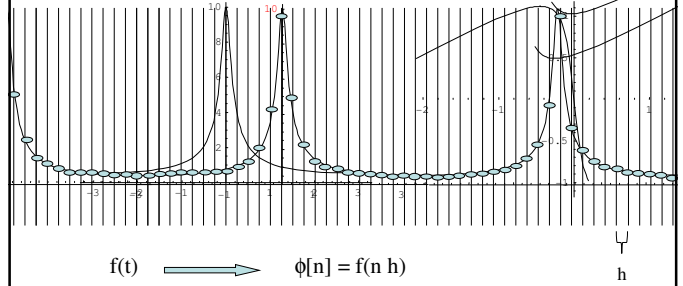
$$\begin{aligned} F[k] &= \frac{1}{p} \int_0^p f(t) e^{-2\pi i k t/p} dt \\ &= \frac{1}{p} \int_0^p f(t) \cos(2\pi k t/p) dt \\ &\quad - i/p \int_0^p f(t) \sin(2\pi k t/p) dt \end{aligned}$$

Inverse Fourier transform $F[k] \rightarrow f(t)$

$$f(t) = \sum_{k \in \mathcal{Z}} F[k] e^{2\pi i k t/p}$$

DFT

$$\int_0^p f(t) e^{-2\pi i k t/p} dt \Rightarrow \sum_{n=0}^{N-1} \phi[n] e^{-2\pi i k n h/p}$$



DFT and its inverse for periodic discrete data

$$\Phi[k] = \sum_{n=0}^{N-1} \phi[n] e^{-2\pi i k n h/p} \quad p = N h$$

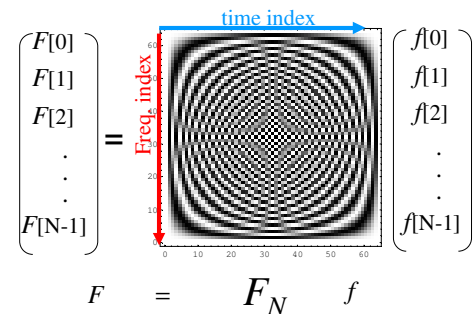
$$= \sum_{n=0}^{N-1} \phi[n] e^{-2\pi i k n/N}$$

This is automatically periodic in k with period N
Inverse is like Fourier series, but with only p terms

Discrete time Numerical Fourier Analysis

DFT is really just a matrix multiplication!

$$F[m] = \frac{1}{N} \sum_{k=0}^{N-1} e^{-2\pi i k m/N} f[k]$$



Fourier Matrices

A Fourier matrix of order n is defined as the following

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

where

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

is an n th root of unity.

$i = \sqrt{-1}$ Primitive Roots of Unity

A number ω is a **primitive n -th root of unity**, for $n > 1$, if

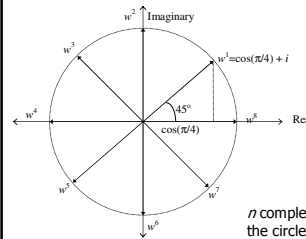
$$\omega^n = 1$$

The numbers $1, \omega, \omega^2, \dots, \omega^{n-1}$ are all distinct

- Example: The complex number $e^{2\pi i/n}$ is a primitive n -th root of unity, where

Check: if properties are satisfied

1. $\omega^j = e^{\frac{2\pi i j}{n}} \neq 1$
2. $\omega^n = \left(e^{\frac{2\pi i}{n}} \right)^n = e^{2\pi i} = \cos 2\pi + i \sin 2\pi = 1$
3. $S = \sum_{j=0}^{n-1} \omega^{nj} = \omega^0 + \omega^1 + \omega^{2j} + \omega^{3j} + \dots + \omega^{j(n-1)} = 0$



n complex roots of unity equally spaced around the circle of unit radius centered at the origin of the complex plane.

Roots of Unity: Properties

- Property 1: Let ω be the principal n th root of unity. If $n > 0$, then $\omega^{n/2} = -1$.
 - Proof: $\omega = e^{2\pi i/n} \Rightarrow \omega^{n/2} = e^{\pi i} = -1$. (Euler's formula)
 - **Reflective Property:**
 - Corollary: $\omega^{k+n/2} = -\omega^k$.
- Property 2: Let $n > 0$ be even, and let ω and v be the principal n th and $(n/2)$ th roots of unity. Then $(\omega^k)^2 = v^k$.
 - Proof: $(\omega^k)^2 = e^{(2k)2\pi i/n} = e^{(k)2\pi i/(n/2)} = v^k$.
 - **Reduction Property:** If ω is a primitive $(2n)$ -th root of unity, then ω^2 is a primitive n -th root of unity.

- L3: Let $n > 0$ be even. Then, the squares of the n complex n th roots of unity are the $n/2$ complex $(n/2)$ th roots of unity.
 - Proof: If we square all of the n th roots of unity, then each $(n/2)$ th root is obtained exactly twice since:
 - L1 $\Rightarrow \omega^{k+n/2} = -\omega^k$
 - thus, $(\omega^{k+n/2})^2 = (\omega^k)^2$
 - L2 \Rightarrow both of these = v^k
 - $\omega^{k+n/2}$ and ω^k have the same square
 - **Inverse Property:** If ω is a primitive root of unity, then $\omega^{-1} = \omega^{n-1}$
 - Proof: $\omega \omega^{n-1} = \omega^n = 1$

Fast Fourier Transform

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

$$\begin{aligned}
 F_k &= \sum_{j=0}^{N-1} e^{2\pi i j k / N} f_j \\
 &= \sum_{j=0}^{N/2-1} e^{2\pi i k(2j) / N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi i k(2j+1) / N} f_{2j+1} \\
 &= \sum_{j=0}^{N/2-1} e^{2\pi i k j / (N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi i k j / (N/2)} f_{2j+1} \\
 &= F_k^e + W^k F_k^o
 \end{aligned}$$

- So far we have seen what happens on the right hand side
- How about the left hand side?
- When we split the sums in two we have two sets of sums with $N/2$ quantities for N points.
- So the complexity is $N^2/2 + N^2/2 = N^2$
- So there is no improvement
- Need to reduce the sums on the right hand side as well
 - We need to reduce the number of sums computed from $2N$ to a lower number
 - Notice that the transforms F_e^k and F_o^k are periodic in k with length $N/2$.
 - So we need only compute half of them!

FFT

- So DFT of order N can be expressed as sum of two DFTs of order $N/2$ evaluated at $N/2$ points
- Does this improve the complexity?
- Yes $(N/2)^2 + (N/2)^2 = N^2/2 < N^2$
- But we are not done ...
- Can apply the lemma recursively

$$F_k^e = F_k^{ee} + W^k F_k^{eo}, \quad F_k^o = F_k^{oe} + W^k F_k^{oo}$$
- Finally we have a set of one point transforms
- One point transform is identity $F_k^{eooo\cdots oee} = f_n$

FFT Algorithm

```

FFT (n, a0, a1, a2, ..., an-1)
if (n == 1) // n is a power of 2
    return a0
ω ← e2πi/n
(e0, e1, e2, ..., en/2-1) ← FFT (n/2, a0, a2, a4, ..., an-2)
(d0, d1, d2, ..., dn/2-1) ← FFT (n/2, a1, a3, a5, ..., an-1)
for k = 0 to n/2 - 1
    yk ← ek + ωk dk
    yk+n/2 ← ek - ωk dk
return (y0, y1, y2, ..., yn-1)
    
```

$$T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$$

Complexity

- Each F_k is a sum of $\log_2 N$ transforms and (factors)
- There are N F_k s
- So the algorithm is $O(N \log_2 N)$
- *This is a recursive algorithm*

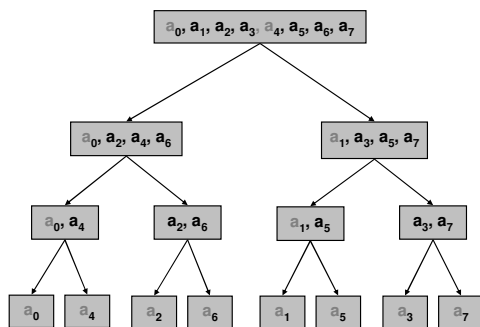
FFTtx

```
function y = ffttx(x)
%FFTtx Textbook Fast Finite
    Fourier Transform.
% FFTTX(X) computes the same
    finite Fourier transform as FFT(X).
% The code uses a recursive divide
    and conquer algorithm for
% even order and matrix-vector
    multiplication for odd order.
% If length(X) is m*p where m is
    odd and p is a power of 2, the
% computational complexity of this
    approach is O(m^2)*O(p*log2(p)).

    if rem(n,2) == 0
        % Recursive divide and conquer
        k = (0:n/2-1)';
        w = omega.^k;
        u = ffttx(x(1:2:n-1));
        v = w.*ffttx(x(2:2:n));
        y = [u+v; u-v];
    else
        % The Fourier matrix.
        j = 0:n-1;
        k = j';
        F = omega.^(k*j);
        y = F*x;
    end

x = x(:);
n = length(x);
omega = exp(-2*pi*i/n);
```

Scrambled Output of the FFT



"bit-reversed" order