Solving Poisson’s Equation

(part 3: with FFT)

Slides for week 10)
Outline

° Poisson equation; various solution strategies

° Traditional (slow) methods like
  • Jacobi’s method
  • Red-Black SOR method
  • Conjugate Gradients

° Hierarchical methods
  • Last week:
    - Multigrid for faster convergence
  • This week:
    - Fast Fourier Transform, or
    - A smart matrix multiplication for special matrices

° Note: Multigrid and FFT not easily applied to general problems
Summary of Jacobi, SOR and CG

° Jacobi, SOR, and CG all perform sparse-matrix-vector multiply

° For Poisson, this means nearest neighbor communication on an \( n \times n \) grid

° It takes \( n = N^{1/2} \) steps for information to travel across an \( n \times n \) grid

° Since solution on one side of grid depends on data on other side of grid faster methods require faster ways to move information
  • Multigrid
  • FFT

° This week:
  • How does FFT manage to travel across grid fast?
Solving the Poisson equation with the (F)FT

° Motivation for Fourier Transform: express continuous solution as a Fourier series
  • \( u(x,y) = \sum_{i} \sum_{k} u_{ik} \sin(i \cdot x) \sin(k \cdot y) \)
  • \( u_{ik} \) are called Fourier coefficients of \( u(x,y) \)

° Poisson’s equation, \( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = b \), becomes

\[ \sum_{i} \sum_{k} (i^2 + k^2) u_{ik} \sin(i \cdot x) \sin(k \cdot y) = \sum_{i} \sum_{k} b_{ik} \sin(i \cdot x) \sin(k \cdot y) \]

° where \( b_{ik} \) are Fourier coefficients of \( b(x,y) \)

° No spatial grid (yet), \( i \) and \( k \) refers to spectral modes
° Summations start at 1 and go on ...
Solving the Poisson equation with the FFT

° By uniqueness of the Fourier series:
  • \( u_{ik} = \frac{b_{ik}}{\sqrt{2(i^2 + k^2)}} \)

° Basic Algorithm (Continuous / Discrete):
  ° Compute Fourier coefficient \( b_{ik} \) of right hand side
    ° Apply 2D FFT to values of \( b(i,k) \) on grid
  ° Compute Fourier coefficients \( u_{ik} \) of solution
    ° Divide each transformed \( b(i,k) \) by \( \text{function}(i,k) \)
  ° Compute solution \( u(x,y) \) from Fourier coefficients
    ° Apply 2D inverse FFT to values of \( b(i,k) \)
Serial Fourier Transform

° Let index of matrices and vectors start from 0
° The *Discrete Fourier Transform (DFT)* of an m-element vector $v$ is:

$$F \ast v$$

Where $F$ is the $m \times m$ matrix defined as:

$$F[j,k] = w^{(j \ast k)}$$

Where

$$w = e^{(2\pi i/m)} = \cos(2\pi/m) + i \ast \sin(2\pi/m)$$

• this is a complex number with whose $m^{th}$ power is 1 and is therefore called the $m^{th}$ root of unity

° E.g., for $m = 4$:  $w = i$, $w^2 = -1$, $w^3 = -i$, $w^4 = 1$

° FFT is the fast evaluation of a DFT
Application of the 1D FFT for filtering

- Signal = \( \sin(7t) + 0.5 \sin(5t) \) at 128 points
- Noise = random number bounded by 0.75
- Filter by zeroing out FFT components < 0.25
Use 2D FFT for image compression

- Image = 200x320 matrix of values
- Compress by keeping largest 2.5% of FFT components
Transforms related to Discrete Fourier Transform

° Most applications require multiplication by both $F$ and inverse($F$).

° Multiplying by $F$ and inverse($F$) are essentially the same. (inverse($F$) is the complex conjugate of $F$ divided by $n$.)

° For solving the Poisson equation and various other applications, we use variations on the FFT
  • The sin transform -- imaginary part of $F$
  • The cos transform -- real part of $F$

° Algorithms are similar, so we will focus on the forward FFT.

° Nothing about FFT yet. How to change a DFT into a FFT?
Serial Algorithm for the FFT

° Compute the FFT of an m-element vector v, F*v

\[(F * v)[j] = \sum_{k=0}^{m-1} F(j,k) * v(k)\]

\[= \sum_{k=0}^{m-1} (j^k) * v(k)\]

\[= \sum_{k=0}^{m-1} (j^k) * v(k)\]

\[= \sum_{k=0}^{m-1} (j^k) * v(k)\]

\[= \sum_{k=0}^{m-1} (j^k) * v(k)\]

° Where V is defined as the polynomial

\[V(x) = \sum_{k=0}^{m-1} x^k * v(k)\]
Divide and Conquer FFT

° V can be evaluated using divide-and-conquer

\[ V(x) = \sum_{k=0}^{m-1} (x)^k \cdot v(k) \]

\[ = v[0] + x^2 \cdot v[2] + x^4 \cdot v[4] + \ldots \]
\[ + x \cdot (v[1] + x^2 \cdot v[3] + x^4 \cdot v[5] + \ldots ) \]
\[ = V_{\text{even}}(x^2) + x \cdot V_{\text{odd}}(x^2) \]

° V has degree m, so \( V_{\text{even}} \) and \( V_{\text{odd}} \) are polynomials of degree \( m/2-1 \)

° Evaluate these at points \((w^2)^j\) for \(0 \leq j \leq m-1\)

° This are just \( m/2 \) different points, since

\[ (w^{j+m/2})^2 = (w^j \cdot w^{m/2})^2 = (w^{2j} \cdot w) = (w^j)^2 \]
Divide-and-Conquer FFT, recursive

\[
\text{FFT}(v, v, m) \\
\text{if } m = 1 \text{ return } v[0] \\
\text{else} \\
\quad v_{\text{even}} = \text{FFT} (v[0:2:m-2], w^2, m/2) \\
\quad v_{\text{odd}} = \text{FFT} (v[1:2:m-1], w^2, m/2) \\
\quad \text{vec} = [w^0, w^1, \ldots, w^{(m/2-1)}] \\
\quad \text{return } [v_{\text{even}} + (\text{vec} \times v_{\text{odd}}), v_{\text{even}} - (\text{vec} \times v_{\text{odd}})]
\]

° The * here is a component-wise multiply!
° [...] is an m-element vector built from 2 m/2-element vectors

This results in an O(m log m) algorithm! Verify!
Iterative or recursive Algorithm

° The call tree of divide and conquer FFT algorithm is a complete binary tree of log m levels

° Iterative start at bottom; Recursive start at top

° Example for 16 elements, i.e., m=4

FTT(0,1,2,3,…,15) = FTT(xxxx)

FFT(1,3,…,15) = FTT(xxx1)

FFT(0,2,…,14) = FTT(xxx0)

FFT(xx10)

FFT(xx11)

FFT(xx00)

FFT(x100)

FFT(x010)

FFT(x110)

FFT(x001)

FFT(x101)

FFT(x011)

FFT(x111)

Each x is 0 or 1

FFT(0)  FFT(8)  FFT(4)  FFT(12)  FFT(2)  FFT(10)  FFT(6)  FFT(14)  FFT(1)  FFT(9)  FFT(5)  FFT(13)  FFT(3)  FFT(11)  FFT(7)  FFT(15)
Iterative FFT Algorithm

° Practical algorithms are iterative, going across each level in the tree starting at the bottom

° Algorithm overwrites \( v[i] \) by \( (F^*v)[\text{bitreverse}(i)] \)

° A final sort is necessary if one is really interested in the Discrete Fourier Transform of the vector \( v \).

° Often this is not required, e.g.,
  • work on transformed (unsorted) data is performed
  • followed by an inverse transform
  • results are again in right sequence
Parallel 1D FFT

° Data dependencies in 1D FFT
  • Butterfly pattern

° A PRAM algorithm takes $O(\log m)$ time
  • each step to right is parallel
  • there are $\log m$ steps

° What about communication cost?

° How to distribute data over processes?
Block Layout of 1D FFT

- Using a block layout (m/p contiguous elements per processor)

- No communication in last log(m) - log(p) steps

- Each step requires fine-grained communication in first log(p) steps
Cyclic Layout of 1D FFT

- Cyclic layout (only 1 element per processor, wrapped). Like dealing cards.

- No communication in first $\log(m) - \log(p)$ steps

- Communication in last $\log(p)$ steps
Parallel Complexity of 1D FFT

- \( m \) = vector size,
- \( p \) = number of processors
- \( f \) = time per flop = 1
- \( a \) = startup for message (in f units)
- \( b \) = time per word in a message (in f units)

\[
\text{Time(blockFFT)} = \text{Time(cyclicFFT)} = \\
\log(m) \times \frac{m}{p} \\
+ \log(p) \times a \\
+ \log(p) \times \frac{m}{p} \times b
\]
Parallel Complexity Analysis

° Time(blockFFT) = Time(cyclicFFT) =
  log(m) * m/p
  + log(p) * a
  + log(p) * m/p * b

° In each of the log(m) stages some work is done on the m/p elements assigned to a process, actually 2 flops per element.

° In log(p) stages each process has to initiate one communication

° In log(p) stages each process has to communicate all its m/p elements to 1 other process.
FFT with “Transpose”

° If we start with a cyclic layout for first \( \log(p) \) steps, there is no communication

° Then transpose the vector for last \( \log(m)-\log(p) \) steps
  - i.e., Move elements to another process

° All communication is now concentrated in the transpose
Why is the communication called Transpose?

- It is analogous to transposing an array
- View as a 2D array of n/p by p
- Note: same idea is useful for uniprocessor caches

<table>
<thead>
<tr>
<th>Processor</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
Complexity of the FFT with Transpose

° If communication is not overlapped:

° Time(transposeFFT) =

\[ \log(m) \times \frac{m}{p} \text{ same as before} + (p-1) \times \frac{1}{p} \times \frac{m}{p} \times \frac{1}{p} \text{ was } \log(p) \times \frac{m}{p} \times \frac{1}{p} \]

° In just one stage each process sends to all p-1 others

° In each of its (p-1) messages, a process sends the fraction 1/p of its m/p elements to another process

° Each process in Transpose version of FFT

• sends less data: factor ~ log(p),
• send more messages: factor ~ p/log(p) compared to block FFT
Complexity of the FFT with Transpose

- If communication can be overlapped
- Assume that all messages can be initiated, without waiting for the “previous” to be established

\[
\text{Time(\text{transposeFFT})} = \log(m) \cdot \frac{m}{p} + 1 \cdot \square + (p-1) \cdot \frac{1}{p} \cdot \frac{m}{p} \cdot \square
\]

- Hence we do not pay for \(p-1\) messages, the second term now becomes simply \(\square\), rather than \((p-1)\square\).
- This is close to optimal.
Higher Dimensional FFTs

° FFTs on 2 or 3 dimensions are defined as 1D FFTs on vectors in all dimensions

° E.g., a 2D FFT first does 1D FFTs on all rows and then all columns

° There are 3 obvious possibilities for the 2D FFT:
  • (1) 2D blocked layout for matrix, using 1D algorithms for each row and column
  • (2) Block row layout for matrix, using serial 1D FFTs on rows, followed by a transpose, then more serial 1D FFTs
  • (3) Block row layout for matrix, using serial 1D FFTs on rows, followed by parallel 1D FFTs on columns
  • Option 1 is best

° For a 3D FFT the options are similar
  • 2 phases done with serial FFTs, followed by a transpose for 3rd
  • can overlap communication with 2nd phase in practice
Summary FFTs

- FFTs basically are smart matrix multiplications
- Their time-complexity equals that of Multigrid methods
- Useful as high performance computational building blocks in several applications, e.g. image processing
- Parallelization based on domain decomposition is not feasible.
  - In each stage any 2 elements being transformed are logically shared by “their” processes
- Data initially on one process effects those on all the others after the \( \log(m) \) stages.
- FFT solution method is NOT iterative like CG or Multigrid