Outline

1. Discrete Fourier Transform
   - Roots of Unity
   - DFT
   - Inverse DFT

2. Convolution
   - Problem

3. Fast Fourier Transform
   - Computing DFT
   - FFT Algorithm

4. Parallel FFT
   - Binary Exchange Parallel FFT
   - Transpose Parallel FFT
For given integer $n$, we use notation

$$\omega_n = \cos(2\pi / n) - i \sin(2\pi / n) = e^{-2\pi i / n}$$

for primitive $n$th root of unity, where $i = \sqrt{-1}$

$n$th roots of unity, sometimes called *twiddle factors* in this context, are then given by $\omega_n^k$ or by $\omega_n^{-k}$, $k = 0, \ldots, n - 1$

For convenience, we will assume that $n$ is power of two, and all logarithms used will be base two

We will also index sequences (components of vectors) starting from 0 rather than 1
**Discrete Fourier Transform**, or **DFT**, of sequence $x = [x_0, \ldots, x_{n-1}]^T$ is sequence $y = [y_0, \ldots, y_{n-1}]^T$ given by

$$y_m = \sum_{k=0}^{n-1} x_k \omega_n^{mk}, \quad m = 0, 1, \ldots, n - 1$$

or

$$y = F_n x$$

where entries of **DFT matrix** $F_n$ are given by

$$\{F_n\}_{mk} = \omega_n^{mk}$$
It is easily seen that

$$F_{n}^{-1} = \frac{1}{n} F_{n}^{H}$$

So inverse DFT is given by

$$x_{k} = \frac{1}{n} \sum_{m=0}^{n-1} y_{m} \omega_{n}^{-mk} \quad k = 0, 1, \ldots, n - 1$$
Example

\[ F_4 = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & \omega^1 & \omega^2 & \omega^3 \\
1 & \omega^2 & \omega^4 & \omega^6 \\
1 & \omega^3 & \omega^6 & \omega^9 \\
\end{bmatrix} = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -i & -1 & i \\
1 & -1 & 1 & -1 \\
1 & i & -1 & -i \\
\end{bmatrix} \]

\[ 4F_4^{-1} = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & \omega^{-1} & \omega^{-2} & \omega^{-3} \\
1 & \omega^{-2} & \omega^{-4} & \omega^{-6} \\
1 & \omega^{-3} & \omega^{-6} & \omega^{-9} \\
\end{bmatrix} = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & i & -1 & -i \\
1 & -1 & 1 & -1 \\
1 & -i & -1 & i \\
\end{bmatrix} \]
Convolution

Convolution takes input \( a \) and \( b \) and computes \( c \)

\[
\forall k \in [0, n - 1] \quad c_k = \sum_{j=0}^{k} a_j b_{k-j}
\]

- If \( a \) and \( b \) are coefficients of degree \( n/2 - 1 \) polynomials

\[
p_a(x) = \sum_{k=0}^{n/2-1} a_k x^k, \quad p_b(x) = \sum_{k=0}^{n/2-1} a_k x^k
\]

the convolution computes the coefficients \( c \) of the product

\[
p_c(x) = p_a(x)p_b(x) = \sum_{k=0}^{n-1} c_k x^k
\]

- naive evaluation costs \( O(n^2) \) operations
Convolution and Toeplitz Matrices

- Convolution can be interpreted as matrix-vector multiplication with a triangular Toeplitz matrix

\[
\begin{bmatrix}
  c_0 & c_1 & c_2 & c_3 \\
  a_1 & a_2 & a_3 & a_4
\end{bmatrix} =
\begin{bmatrix}
  b_0 & b_1 & b_2 & b_3 \\
  0 & b_0 & b_1 & b_2 \\
  0 & 0 & b_0 & b_1 \\
  0 & 0 & 0 & b_0
\end{bmatrix}
\]

- Toeplitz and Hankel matrices (in the latter, each antidiagonal is defined by a single element) provide a general matrix representation for convolutional operators.
Convolution via Interpolation by DFT

- The DFT, $F_n a$ evaluates polynomial $p_a$ at each $\omega^j$
- The values of $p_c$ at each $\omega^j$ are then easily obtained

$$p_c(\omega^j) = p_a(\omega^j)p_b(\omega^j)$$

- The inverse DFT, $F_n^{-1} p_c(x)$ interpolates the values of the polynomial $p_c$ at each $\omega^j$ producing its coefficients $c$
- The overall procedure is described by

$$c = F_n^{-1} [(F_n a) \odot (F_n b)]$$

where $\odot$ is an elementwise product ($a$ and $b$ are padded with trailing zeros)
Convolution via DFT

- Lets write out the full expression

\[ c_k = \frac{1}{n} \sum_s \omega_n^{-ks} \left( \sum_j \omega_n^{sj} a_j \right) \left( \sum_t \omega_n^{st} b_t \right) \]

- Rearrange the order of the summations to see what happens to every product of \( a \) and \( b \)

\[ c_k = \frac{1}{n} \sum_s \sum_j \sum_t \omega_n^{(j+t-k)s} a_j b_t \]

- For any \( u = j + t - k \neq 0 \), we observe \( \sum_s (\omega_n^u)^s = 0 \)

- When \( j + t - k = 0 \) the products \( \omega_n^{(s+t-j)k} = 1 \), so there are \( n \) nonzero terms \( a_j b_{k-j} \) in the summation
Computing DFT

- To illustrate, consider computing DFT for \( n = 4 \),

\[
y_m = \sum_{k=0}^{3} x_k \omega_n^{mk}, \quad m = 0, \ldots, 3
\]

- Writing out equations in full,

\[
\begin{align*}
y_0 &= x_0 \omega_n^0 + x_1 \omega_n^0 + x_2 \omega_n^0 + x_3 \omega_n^0 \\
y_1 &= x_0 \omega_n^0 + x_1 \omega_n^1 + x_2 \omega_n^2 + x_3 \omega_n^3 \\
y_2 &= x_0 \omega_n^0 + x_1 \omega_n^2 + x_2 \omega_n^4 + x_3 \omega_n^6 \\
y_3 &= x_0 \omega_n^0 + x_1 \omega_n^3 + x_2 \omega_n^6 + x_3 \omega_n^9
\end{align*}
\]
Noting that

\[ \omega_0^n = \omega_4^n = 1, \quad \omega_2^n = \omega_6^n = -1, \quad \omega_9^n = \omega_1^n \]

and regrouping, we obtain

\[
\begin{align*}
y_0 &= (x_0 + \omega_0^n x_2) + \omega_0^n (x_1 + \omega_0^n x_3) \\
y_1 &= (x_0 - \omega_0^n x_2) + \omega_1^n (x_1 - \omega_0^n x_3) \\
y_2 &= (x_0 + \omega_0^n x_2) + \omega_2^n (x_1 + \omega_0^n x_3) \\
y_3 &= (x_0 - \omega_0^n x_2) + \omega_3^n (x_1 - \omega_0^n x_3)
\end{align*}
\]

DFT can now be computed with only 8 additions and 6 multiplications, instead of expected \((4 - 1) \times 4 = 12\) additions and \(4^2 = 16\) multiplications.
Actually, even fewer multiplications are required for this small case, since $\omega_n^0 = 1$, but we have tried to illustrate how algorithm works in general.

Main point is that computing DFT of original 4-point sequence has been reduced to computing DFT of its two 2-point even and odd subsequences.

This property holds in general: DFT of $n$-point sequence can be computed by breaking it into two DFTs of half length, provided $n$ is even.
General pattern becomes clearer when viewed in terms of first few Fourier matrices

\[ F_1 = 1, \quad F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \]

Let \( P_4 \) be permutation matrix

\[ P_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
Computing DFT

- Let $F_2$ be diagonal matrix
  \[
  F_2 = \text{diag}(1, \omega_4) = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}
  \]

- Then we have
  \[
  F_4 P_4 = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -1 & -i & i \\
1 & 1 & -1 & -1 \\
1 & -1 & i & -i \\
  \end{bmatrix} = \begin{bmatrix} F_2 & F_2 F_2 \\ F_2 & -F_2 F_2 \end{bmatrix}
  \]

- Thus, $F_4$ can be rearranged so that each block is diagonally scaled version of $F_2$.

- Such hierarchical splitting can be carried out at each level, provided number of points is even.
In general, $P_n$ is permutation that groups even-numbered columns of $F_n$ before odd-numbered columns, and

$$F_{n/2} = \text{diag}(1, \omega_n, \ldots, \omega_{n/2}^{(n/2)-1})$$

To apply $F_n$ to sequence of length $n$, we need merely apply $F_{n/2}$ to its even and odd subsequences and scale results, where necessary, by $\pm F_{n/2}$

Resulting recursive divide-and-conquer algorithm for computing DFT is called \textit{Fast Fourier Transform}, or \textit{FFT}

FFT is particular way of computing DFT efficiently
Consider $b = F_n a$, we have

$$\forall j \in [0, n - 1] \quad b_j = \sum_{k=0}^{n-1} \omega_n^{jk} a_k$$

Express DFT as two DFTs of dimension $n/2$, with a different root of unity $\omega_{n/2}$

Separate summands into odds and evens, use $\omega_{n/2} = \omega_n^2$

\[
b_j = \sum_{k=0}^{n/2-1} \omega_n^{j(2k)} a_{2k} + \sum_{k=0}^{n/2-1} \omega_n^{j(2k+1)} a_{2k+1}
\]

\[
= \sum_{k=0}^{n/2-1} \omega_n^{jk} a_{2k} + \omega_n^j \sum_{k=0}^{n/2-1} \omega_n^{jk} a_{2k+1}
\]
Radix-2 Fast Fourier Transform (FFT), contd.

\[ b_j = \sum_{k=0}^{n/2-1} \omega_{n/2}^{jk} a_{2k} + \omega_n^j \sum_{k=0}^{n/2-1} \omega_{n/2}^{jk} a_{2k+1} \]

The summations for \( b_j \) and \( b_{j+n/2} \) are closely related,

\[ b_{j+n/2} = \sum_{k=0}^{n/2-1} \omega_n^{(j+n/2)k} a_{2k} + \omega_n^{j+n/2} \sum_{k=0}^{n/2-1} \omega_n^{(j+n/2)k} a_{2k+1} \]

Now \( \omega_n^{(j+n/2)k} = \omega_n^{jk} \) since \( (\omega_n^{n/2})^k = 1^k = 1 \) and using \( \omega_n^{n/2} = -1 \),

\[ b_{j+n/2} = \sum_{k=0}^{n/2-1} \omega_n^{jk} a_{2k} - \omega_n^j \sum_{k=0}^{n/2-1} \omega_n^{jk} a_{2k+1} \]
Radix-2 Fast Fourier Transform (FFT), contd.

- Let vectors $u$ and $v$ be two recursive FFTs, $\forall j \in [0, n/2 - 1]$
  \[
  u_j = \sum_{k=0}^{n/2-1} \omega_{n/2}^{jk} a_{2k}, \quad v_j = \sum_{k=0}^{n/2-1} \omega_{n/2}^{jk} a_{2k+1}
  \]

- Given $u$ and $v$ scale using "twiddle factors" $z_j = \omega_n^j \cdot v_j$

- Then it suffices to combine the vectors as follows $b = \begin{bmatrix} u + z \\ u - z \end{bmatrix}$

- This recombination is an FFT of dimension 2
  \[
  b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \text{vec} \left( \begin{bmatrix} b_1 & b_2 \end{bmatrix} \right) = \text{vec} \left( \begin{bmatrix} u & z \\ 1 & 1 \\ 1 & -1 \end{bmatrix} \right)
  \]

- Radix-$r$ algorithm for any $A \in \mathbb{R}^{n/r \times r}$
  \[
  \text{vec} \left( B \right) = F_n \text{vec} \left( A \right) \quad \text{if and only if} \quad B = F_{n/r} A F_r
  \]
procedure \texttt{fft}(x, y, n, \omega) \\
\textbf{if} \ n = 1 \ \textbf{then} \\
\quad y[0] = x[0] \\
\textbf{else} \\
\quad \textbf{for} \ k = 0 \ \textbf{to} \ (n/2) - 1 \\
\quad \quad p[k] = x[2k] \\
\quad \quad s[k] = x[2k + 1] \\
\quad \textbf{end} \\
\quad \texttt{fft}(p, q, n/2, \omega^2) \\
\quad \texttt{fft}(s, t, n/2, \omega^2) \\
\quad \textbf{for} \ k = 0 \ \textbf{to} \ n - 1 \\
\quad \quad y[k] = q[k \mod (n/2)] + \omega^k t[k \mod (n/2)] \\
\quad \textbf{end} \\
\textbf{end}
Complexity of FFT Algorithm

- There are $\log n$ levels of recursion, each of which involves $\Theta(n)$ arithmetic operations, so total cost is $\Theta(n \log n)$.

- By contrast, straightforward evaluation of matrix-vector product defining DFT requires $\Theta(n^2)$ arithmetic operations, which is enormously greater for long sequences.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>384</td>
<td>4096</td>
</tr>
<tr>
<td>128</td>
<td>896</td>
<td>16384</td>
</tr>
<tr>
<td>256</td>
<td>2048</td>
<td>65536</td>
</tr>
<tr>
<td>512</td>
<td>4608</td>
<td>262144</td>
</tr>
<tr>
<td>1024</td>
<td>10240</td>
<td>1048576</td>
</tr>
</tbody>
</table>
FFT Algorithm

- For clarity, separate arrays were used for subsequences, but transform can be computed in place using no additional storage.

- Input sequence is assumed complex; if input sequence is real, then additional symmetries in DFT can be exploited to reduce storage and operation count by half.

- Output sequence is not produced in natural order, but either input or output sequence can be rearranged at cost of $\Theta(n \log n)$, analogous to sorting.

- FFT algorithm can be formulated using iteration rather than recursion, which is often desirable for greater efficiency or when programming language does not support recursion.
Because of similar form of DFT and its inverse, FFT algorithm can also be used to compute inverse DFT efficiently.

Ability to transform back and forth quickly between time and frequency domains makes it practical to perform any computations or analysis that may be required in whichever domain is more convenient and efficient.
To obtain fine-grain decomposition of FFT, we assign input data $x_k$ to task $k$, which also computes result $y_k$.

At stage $m$ of algorithm, tasks $k$ and $j$ exchange data, where $k$ and $j$ differ only in their $m$th bits.
There are $n$ tasks and $\log n$ stages, so parallel time required to compute FFT is

$$T_n = (\gamma + \alpha + \beta) \log n$$

where $\gamma$ is cost of multiply-add, and $\alpha + \beta$ is cost of exchanging one number between pair of tasks at each stage

Hypercube is natural network for FFT algorithm
To obtain smaller number of coarse-grain tasks, agglomerate sets of $n/p$ components of input and output vectors $x$ and $y$, where we assume $p$ is also power of two.
Components having their $\log p$ most significant bits in common are assigned to the same task.

Thus, exchanges are required in binary exchange algorithm only for first $\log p$ stages, since data are local for remaining $\log(n/p)$ stages.
Each stage involves updating of $n/p$ components by each task, and exchange of $n/p$ components for each of first $\log p$ stages.

Thus, total time required using hypercube network is

$$T_p = \alpha (\log p) + \beta n (\log p)/p + \gamma n (\log n)/p$$

To determine isoefficiency function, set

$$\gamma n \log n \approx E(\alpha p \log p + \beta n \log p + \gamma n \log n)$$

which holds if $n = \Theta(p)$, so isoefficiency function is $\Theta(p \log p)$, since $T_1 = \Theta(n \log n)$
Transpose Parallel FFT

- Binary exchange algorithm has one phase that is communication free and another phase that requires communication at each stage.

- Another approach is to realign data so that both computational phases are communication free, and only communication is for data realignment phase between computational phases.

- To accomplish this, data can be organized in $\sqrt{n} \times \sqrt{n}$ array, as illustrated next for $n = 16$. 
Transpose Parallel FFT

Initial phase

Data realignment phase

Final phase

Transpose
If array is partitioned by columns, which are assigned to $p \leq \sqrt{n}$ tasks, then no communication is required for first $\log(\sqrt{n})$ stages.

Data are then transposed using all-to-all personalized collective communication, so that each row of data array is now stored in single task.

Thus, final $\log(\sqrt{n})$ stages now require no communication.

Overall performance of transpose algorithm depends on particular implementation of all-to-all personalized collective communication.
Transpose Parallel FFT

- Straightforward approach yields total parallel time

\[ T_p = \alpha \log p + \beta n \log p/p + \gamma n (\log n)/p \]

- Compared with binary exchange algorithm, transpose algorithm has higher cost due to message start-up but lower cost due to per-word transfer time

- Thus, choice of algorithm depends on relative values of \( \alpha \) and \( \beta \) for given parallel system
References


References

- P. N. Swarztrauber, Multiprocessor FFTs, *Parallel Computing* 5:197-210, 1987