Fast Fourier Transform

Přemysl Šůcha

Topic Overview

• Introduction to Fast Fourier Transform
• Binary-Exchange algorithm
• Transpose algorithm
Introduction to Fast Fourier Transform

- The discrete Fourier transform (DFT) plays an important role in many applications, including digital signal processing, image filtering, solutions to linear partial differential equations, convolution …
- The DFT is a linear transformation that maps \( n \) regularly sampled points from a cycle of a periodic signal onto an equal number of points representing the frequency spectrum of the signal.
- In 1965, Cooley and Tukey devised an algorithm to compute the DFT of an \( n \)-point series in \( O(n \log n) \) operations. Its variations are referred to as the Fast Fourier Transform (FFT).
The Serial Algorithm

• Consider a sequence $X = <X[0], X[1], ..., X[n-1]>$ of length $n$. The discrete Fourier transform of the sequence $X$ is the sequence $Y = <Y[0], Y[1], ..., Y[n-1]>$, where

$$Y[l] = \sum_{k=0}^{n-1} X[k] \omega^{kl}, \ 0 \leq l < n$$

• $\omega$ is the $n$-th root of unity in the complex plane; that is $\omega = e^{2\pi\sqrt{-1}/n}$. 
The Serial Algorithm

• The powers of $\omega$ used in an FFT computation are also known as twiddle factors.

• Note: $\omega = e^{2\pi \sqrt{-1}/n} = \cos \left(\frac{2\pi}{n}\right) + i \sin \left(\frac{2\pi}{n}\right)$

• Power of roots of unity are periodic with period $n$. 
The Serial Algorithm

- The **sequential complexity** of computing the entire sequence Y of length n is $O(n^2)$.
- The **fast Fourier transform** algorithm reduces this complexity to $O(n \log n)$.
- The FFT algorithm is based on the idea that permits an $n$-point DFT computation to be **split into two $(n/2)$-point DFT** computations.
Two-point DFT (n=2)

• For $n=2$ the twiddle factor is $\omega = e^{2\pi \sqrt{-1}/n} = e^{-\pi i} = -1$.
• Then DFT is:

$$Y[l] = \sum_{k=0}^{n-1} X[k] (-1)^{kl} = X[0](-1)^{l0} + X[1](-1)^{l1} = X[0] + X[1](-1)^l$$

so

$$Y[0] = X[0] + X[1] \quad \text{and} \quad Y[1] = X[0] - X[1]$$

(* $e^{i\theta} = \cos \theta + i \sin \theta$)
Four-point DFT (n=4)

- For $n=4$ the twiddle factor is $\omega = e^{-i\pi/2} = -i$.
- Then DFT is:

$$ Y[l] = \sum_{k=0}^{n-1} X[k] (-i)^{kl} = $$

$$ = X[0] + X[1](-i)^l + X[2](-1)^l + X[3]i^l $$

So


Four-point DFT (n=4)

- To compute Y faster, we can precompute common subexpressions:

\[
\begin{align*}
Y[0] &= (X[0] + X[2]) + (X[1] + X[3]), \\
\end{align*}
\]

- Pre-computation of brackets in two-point DFT can save a lot of addition operations.
If \( n \) is a power of two (e.g. 8 in the figure above), each of these DFT computations can be divided similarly into smaller computations in a recursive manner. This leads to the recursive FFT algorithm.
The Serial Algorithm

- This FFT algorithm is called the **radix-2 algorithm** because at each level of recursion, the input sequence is split into two equal halves.

```plaintext
1. procedure R_FFT(X, Y, n, w)
2. if (n = 1) then Y[0] := X[0] else
3. begin
4.   R_FFT(<X[0], X[2], ..., X[n - 2]>, <Q[0], Q[1], ..., Q[n/2]>, n/2, w^2);
5.   R_FFT(<X[1], X[3], ..., X[n - 1]>, <T[0], T[1], ..., T[n/2]>, n/2, w^2);
6.   for i := 0 to n - 1 do
7.     Y[i] := Q[i mod (n/2)] + w^i T[i mod (n/2)];
8. end R_FFT
```
The Serial Algorithm

• The maximum **number of levels of recursion** is $\log n$ for an initial sequence of length $n$.
• The total number of **arithmetic operations** (line 7) at each level is $O(n)$.
• The overall **sequential complexity** of the algorithm is $O(n \log n)$.

• The serial FFT algorithm can also be cast in an **iterative form**.
• An iterative FFT algorithm is derived by casting each level of recursion, starting with the deepest level, as an iteration.
Cooley-Tukey algorithm

- The **outer loop** (line 5) is executed $\log n$ times for an $n$-point FFT, and the **inner loop** (line 8) is executed $n$ times during each iteration of the outer loop.

```plaintext
1. procedure ITERATIVE_FFT(X, Y, n)
2. begin
3.     r := log n;
4.     for i := 0 to n - 1 do R[i] := X[i];
5.     for m := 0 to r - 1 do /* Outer loop */
6.         begin
7.             for i := 0 to n - 1 do S[i] := R[i];
8.             for i := 0 to n - 1 do /* Inner loop */
9.                 begin
10.                    /* Let (b0b1 ... br -1) be the binary representation of i */
11.                        j := (b0...bm-1 0 bm+1...br -1);
12.                        k := (b0...bm-1 1 bm+1...br -1);
13.                        R[i] := S[j] + S[k] x $\omega^{(bm, bm-1, b0, 0, ..., 0)}$;
14.                    endfor; /* Inner loop */
15.                 endfor; /* Outer loop */
16.     for i := 0 to n - 1 do Y[i] := R[i];
17. end ITERATIVE_FFT
```
The pattern of combination of elements of the input and the intermediate sequences during a 16-point unordered FFT computation.
Binary-Exchange algorithm

• The decomposition for the parallel algorithm is induced by partitioning the input or the output vector.
• We first consider a simple mapping in which one task is assigned to each process.
• Each task starts with one element of the input vector and computes the corresponding element of the output. Process $i$ ($0 \leq i < n$) initially stores $X[i]$ and finally generates $Y[i]$.
• In each of the log $n$ iterations of the outer loop, process $P_i$ updates the value of $R[i]$ by executing line 12 of Cooley-Tukey algorithm.
• All $n$ updates are performed in parallel.
16-point FFT on 16 processes

Parallel mapping where one task is assigned to each process.
Binary-Exchange algorithm

• To perform the updates, process $P_i$ requires an element of $S$ from a process whose label differs from $i$ at only one bit.

• Parallel FFT computation maps naturally onto a hypercube with a one-to-one mapping of processes to nodes.

• In each of the log $n$ iterations of this algorithm, every process performs one complex multiplication and addition, and exchanges one complex number with another process.

• Now consider a mapping in which the $n$ are mapped onto $p$ processes.
Binary-Exchange algorithm

- For the sake of simplicity, let us assume that both $n$ and $p$ are powers of two, i.e., $n = 2^r$ and $p = 2^d$.
- Elements with indices differing at their $d$ (= 2) most significant bits are mapped onto different processes, i.e. there is no interaction during the last $r - d$ iterations.
- Each interaction operation exchanges $n/p$ words of data. The time spent in communication in the entire algorithm is $t_s \log p + t_w(n/p) \log p$.
- The parallel run time of the algorithm on a $p$-node hypercube network is

$$T_P = t_c \frac{n}{p} \log n + t_s \log p + t_w \frac{n}{p} \log p.$$
Transpose Algorithm

• The binary-exchange algorithm yields good performance on parallel computers with sufficiently high communication bandwidth with respect to the processing speed of the CPUs.

• The simplest transpose algorithm requires a single transpose operation over a two-dimensional array; we call this algorithm the two-dimensional transpose algorithm.

• Assume that $\sqrt{n}$ is a power of 2, and that the input sequence of size $n$ is arranged in a $\sqrt{n} \times \sqrt{n}$ two-dimensional square array.
The pattern of combination of elements in a 16-point FFT when the data are arranged in a 4 x 4 two-dimensional square array.
Transpose Algorithm

• The FFT computation in each column can proceed independently for $\log \sqrt{n}$ iterations without any column requiring data from any other column.

• Similarly, in the remaining $\log \sqrt{n}$ iterations, computation proceeds independently in each row without any row requiring data from any other row.
Two-dimensional (2D) transpose

The 2D transpose algorithm for a 16-point FFT on four processes.
Transpose Algorithm (n>p)

- The 2D array of data is striped into blocks, and one block of $\sqrt{n}/p$ rows is assigned to each process.
- In the **first and third phases** of the algorithm, each process computes $\sqrt{n}/p$ FFTs of $\sqrt{n}$ each.
- The **second phase** transposes the $\sqrt{n} \times \sqrt{n}$ matrix (all-to-all personalized communication).
- The parallel run time of the **transpose** algorithm on a **hypercube** is:

$$T_P = 2t_c \frac{\sqrt{n}}{p} \sqrt{n} \log \sqrt{n} + t_s (p - 1) + t_w \frac{n}{p}$$

$$= t_c \frac{n}{p} \log n + t_s (p - 1) + t_w \frac{n}{p}$$
Binary-Exchange vs. Transpose Algorithm

- Parallel runtime of the transpose algorithm
  \[ T_P = t_c \frac{n}{p} \log n + t_s (p - 1) + t_w \frac{n}{p} \]
  has a much higher overhead than the binary-exchange algorithm
  \[ T_P = t_c \frac{n}{p} \log n + t_s \log p + t_w \frac{n}{p} \log p. \]
  due to the message startup time \( t_s \), but has a lower overhead due to per-word transfer time \( t_w \).

- If the latency \( t_s \) is very low, then the transpose algorithm may be the algorithm of choice.
- The binary-exchange algorithm may perform better on a parallel computer with a high communication bandwidth but a significant startup time.