CSE 6220/CX 4220
Introduction to HPC

Lecture 17: Fast Fourier Transform

Helen Xu
hxu615@gatech.edu

Slide credits mostly to Prof. Srinivas Aluru

Motivation

- Listed as one of the top 10 algorithms of the 20th century
- Frequently used in scientific computing

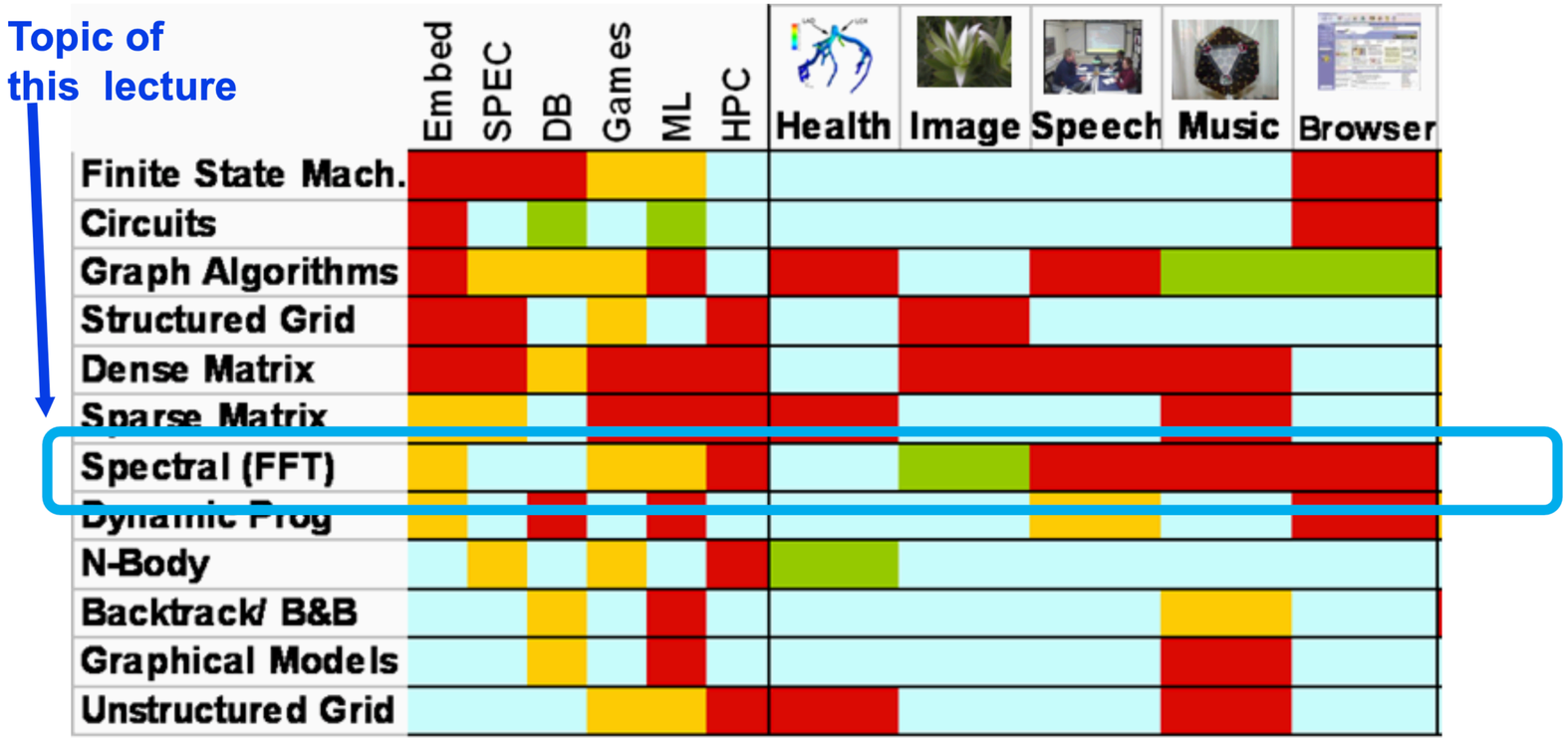


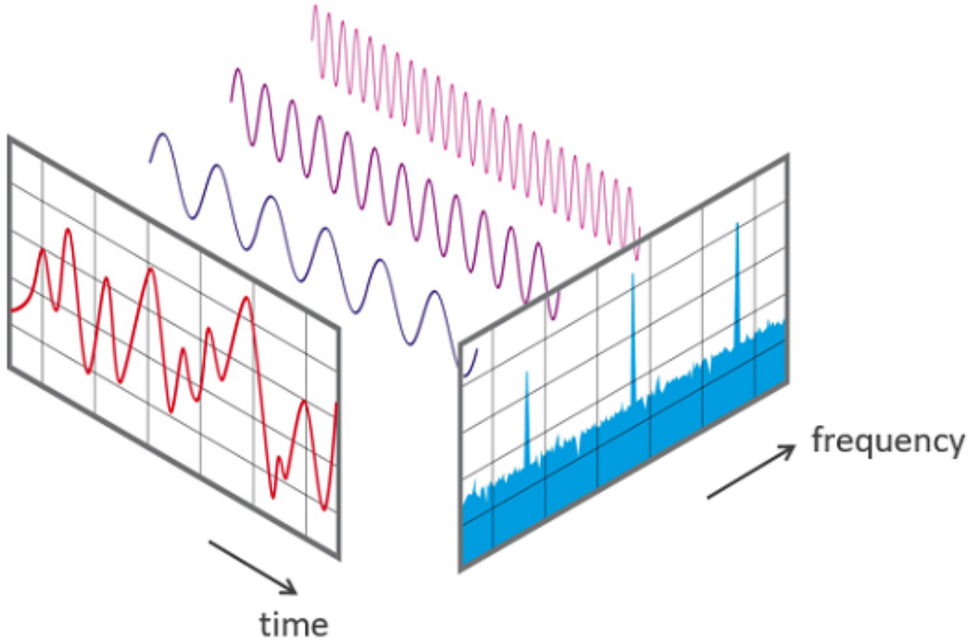
diagram from UC Berkeley CS267

from “The Landscape of Parallel Computing Research: A View from Berkeley” by Asanović et al. ’06

Intuition about Discrete Fourier Transform

- Comes from converting a signal from an original domain (e.g., time or space) to a representation in the frequency domain

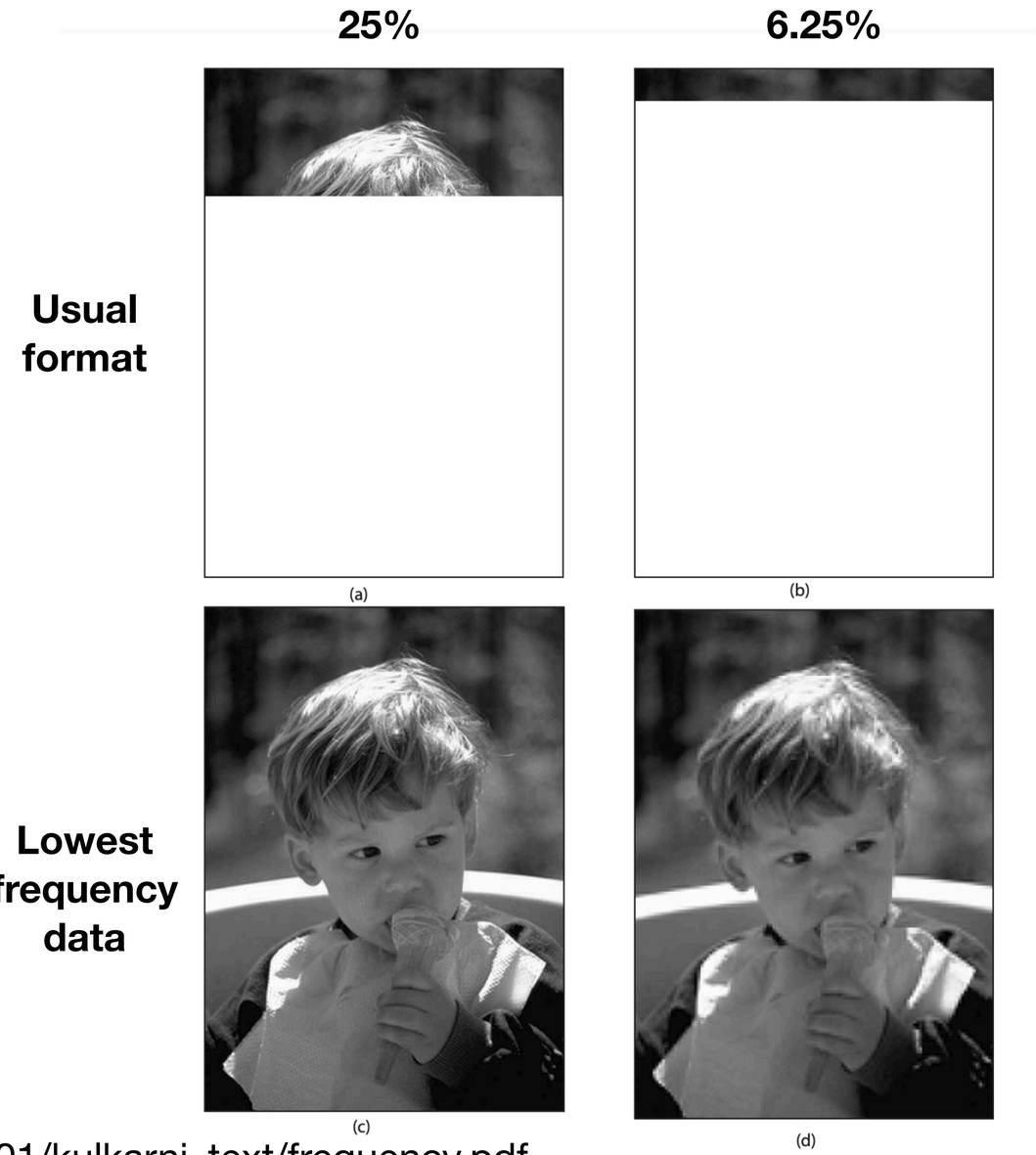
- Components are single sinusoidal oscillations at different frequencies (each with their own amplitude and phase)



https://www.nti-audio.com/en/support/know-how/fast-fourier-transform-fft

Application: Image Representation

- Idea: Represent images without all of the original pixels to save space / time
- Represent “coarse” (low frequency) information separately from “fine” (high frequency) information



https://www.princeton.edu/~cuff/ele201/kulkarni_text/frequency.pdf

Discrete Fourier Transform

$$\begin{bmatrix} P(\omega^0) \\ P(\omega^1) \\ P(\omega^2) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^1 & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

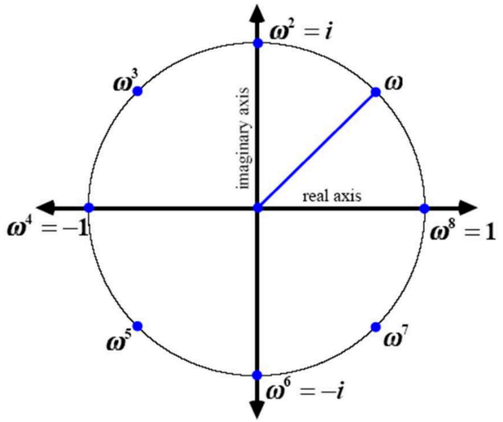
$$P(\vec{\omega}) = M(\omega)\vec{a}$$

$O(n^2)$ time using matrix-vector multiply

DFT: Discrete Fourier Transform – Multiplying given $n \times 1$ vector $\vec{a}$ with the special $n \times n$ matrix $M(\omega)$ given by $M[i,j] = \omega^{ij}$ where ω is the primitive n^{th} root of 1.

DFT is an important numerical method with applications in signal processing, image processing, etc.

Roots of Unity



- A root of unity is the n-th root of 1 for some value of n.
- That is, a complex number ω is an n-th root of unity if $\omega^n = 1$
- There are n complex n-th roots of unity, each of which can be written:

$$e^{\frac{2\pi ik}{n}} \text{ for } k = 0,1,\ldots,n-1 \text{ and } i = \sqrt{-1}$$

- The roots of unity can all be defined as powers of the k=1st one, which is the **primitive n-th root of unity**.

ω is a primitive n-th root of unity if $\omega^n = 1$ and $\omega^j \neq 1$ for $0 < j < n$

Properties of Roots of Unity

P1: Let ω be the primitive n-th root of unity. If $n > 0$, then $\omega^{n/2} = -1$

- Proof: $\omega = e^{\frac{2\pi i}{n}} \Rightarrow \omega^{n/2} = e^{\pi i} = -1$

P2: Let $n > 0$ be even, and let ω and v be the primitive n-th root and (n/2)-th roots of unity. Then $(\omega^k)^2 = v^k$

- Proof: $(\omega^k)^2 = e^{(2k)2\pi i/n} = e^{(k)2\pi i/(n/2)} = v^k$

Properties of Roots of Unity

P3: 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:
- From P1, we have $\omega^{k+n/2} = -\omega^k$, so $(\omega^{k+n/2})^2 = (\omega^k)^2$
- From P2, both of those are v^k, so $\omega^k, \omega^{k+n/2}$ have the same square

Discrete Fourier Transform

$O(n^2)$ time using matrix-vector multiply

$$\begin{bmatrix} P(\omega^0) \\ P(\omega^1) \\ P(\omega^2) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^1 & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

$$P(\vec{\omega}) = M(\omega)\vec{a}$$

DFT: Discrete Fourier Transform – Multiplying given $n \times 1$ vector $\vec{a}$ with the special $n \times n$ matrix $M(\omega)$ given by $M[i,j] = \omega^{ij}$ where ω is the primitive n^{th} root of 1.

DFT is an important numerical method with applications in signal processing, image processing, etc.

FFT: Fast Fourier Transform – Cooley and Tukey's fast polynomial evaluation based algorithm to compute the matrix vector product in $O(n \log n)$ time.

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$$

$O(n \lg n)$ time using FFT

Polynomial view of DFT

- The output vector can be viewed as the evaluation of a polynomial at different powers of ω

- FFT algorithms are based on this polynomial view (not matrix-vector product)

$$\begin{bmatrix} P(\omega^0) \\ P(\omega^1) \\ P(\omega^2) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^1 & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$$

Computing n of these - initially not helping as it takes O(n) time to evaluate each one

Polynomial Multiplication

- $P(x)$ is a polynomial of degree $m-1$

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{m-1} x^{m-1}$$

- $Q(x)$ is a polynomial of degree $n-1$

$$Q(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_{n-1} x^{n-1}$$

- Compute $R(x) = P(x) \times Q(x)$

Notes:

- $R(x)$ is a polynomial of degree $m+n-2$.
- Its $m+n-1$ coefficients $c_0, c_1, \ldots c_{m+n-2}$ are given by

$$c_i = \sum_{j=max\{0,i-n+1\}}^{min\{i,m-1\}} a_j \times b_{i-j}$$

- Serial run-time $= O(mn)$. Can we do better?

Value-based Polynomial Representation

Let $P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{m-1} x^{m-1}$

- Suppose we know $P(x_0), P(x_1), \ldots, P(x_{m-1})$ for m distinct values of x.
- We can write m equations of the form

$$P(x_i) = a_0 + a_1 x_i + a_2 x_i^2 + \ldots + a_{m-1} x_i^{m-1}$$

- Solving the m equations allows us to find coefficients.
  $\Rightarrow$ A polynomial of degree $m-1$ can be uniquely represented by its value at m distinct points.

Multiplication in Value-Based Representation

- $R(x) = P(x) \times Q(x)$
- Suppose we know $P(x)$ and $Q(x)$ at $m + n - 1$ distinct values.
- We can compute $R(x)$ at $m + n - 1$ distinct values through number multiplication.

$$R(x_i) = P(x_i) \times Q(x_i)$$

- Polynomials can be multiplied in $O(m + n)$ time!

Is this fair?

- Value based representation is not suitable for polynomial evaluation. We can't assume a different representation at will.
- Coefficient based representation is natural to input polynomials.
- To specify $P(x)$, we only need its value at m distinct values of x. How can we assume its value is known at $m + n - 1$ distinct values, based on another polynomial of degree $n - 1$?

Strategy Based on Value Representation

Assume both input and output polynomials must be in coefficient based representation.

Algorithm:

Step 1 Evaluate $P(x)$ and $Q(x)$ at $m+n-1$ distinct values of x.
i.e., convert $P(x)$ and $Q(x)$ to value-based representation.
$O((m+n) \cdot m + (m+n) \cdot n) = O((m+n))^2$ time.

Step 2 Carry out polynomial multiplication in value-based representation.
$O(m+n)$ time. Nice!

Step 3 Convert $R(x)$ back to coefficient based representation.
Solve $m+n-1$ equations in as many unknowns.
$O((m+n)^3)$ time!

Run-time just went up from quadratic to cubic!

Key Idea: We have a choice in choosing the distinct values of x.

Cooley-Tukey Algorithm: Choosing Values to Reduce Computation

w.l.o.g. assume an n coefficient polynomial needs to be evaluated at n distinct values. Assume n is a power of 2.

$$P(x) = \sum_{i=0}^{n-1} a_i x^i$$

Idea: Choose pairs of values $x = \alpha$ and $x = -\alpha$

$$P(\alpha) = \sum_{i=0}^{\frac{n}{2}-1} a_{2i}\alpha^{2i} + \alpha \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1}\alpha^{2i}$$

Overlapping terms

$$P(-\alpha) = \sum_{i=0}^{\frac{n}{2}-1} a_{2i}\alpha^{2i} - \alpha \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1}\alpha^{2i}$$

Cooley-Tukey Algorithm: Choosing Values to Reduce Computation

Let

$$P_e(x) = \sum_{i=0}^{\frac{n}{2}-1} a_{2i} x^i$$

$$P_o(x) = \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} x^i$$

Then

$$P(\alpha) = P_e(\alpha^2) + \alpha P_o(\alpha^2)$$

$$P(-\alpha) = P_e(\alpha^2) - \alpha P_o(\alpha^2)$$

Strategy:

- Choose $x_0, x_1, \ldots x_{n-1}$ such that $x_{i+\frac{n}{2}} = -x_i$
- Evaluating n-coefficient polynomial at n distinct values $\Rightarrow$ Evaluating two $\frac{n}{2}$-coefficient polynomials at $\frac{n}{2}$ distinct values

Runtime is given by 2T(n/2) + O(n)

How to apply the strategy recursively

We need:

$$x_{i+\frac{n}{2}} = -x_i \qquad \forall 0 \leq i < \frac{n}{2}$$

$$x^2_{i+\frac{n}{4}} = -x^2_i \qquad \forall 0 \leq i < \frac{n}{4}$$

$$x^4_{i+\frac{n}{8}} = -x^4_i \qquad \forall 0 \leq i < \frac{n}{8}$$

$$\vdots$$

$$x^{2^{k-1}}_{i+\frac{n}{2^k}} = -x^{2^{k-1}}_i \qquad \forall 0 \leq i < \frac{n}{2^k}$$

But How?

Key Idea: Choose complex numbers.

$x_i = \omega^i$, where $\omega = e^{\frac{2\pi j}{n}}$ and $j = \sqrt{-1}$

Roots of unity have the desired property

Lemma

Lemma: If $x_i = \omega^i$ where ω is the n^{th} root of 1, then

$$x_{i+\frac{n}{2^k}}^{2^{k-1}} = -(x_i)^{2^{k-1}}$$

Second half is the same but with the sign flipped

Proof:

$$x_{i+\frac{n}{2^k}}^{2^{k-1}} = \left(\omega^{i+\frac{n}{2^k}}\right)^{2^{k-1}} = \left(\left(e^{\frac{2\pi j}{n}}\right)^{i+\frac{n}{2^k}}\right)^{2^{k-1}}$$

$$e^{\pi j + \frac{2^k \pi j i}{n}} = e^{\pi j} e^{\frac{2^k \pi j i}{n}} = -\left(e^{\frac{2\pi j i}{n}}\right)^{2^{k-1}}$$

$$= -\left(\omega^i\right)^{2^{k-1}} = -x_i^{2^{k-1}}$$

Why does recursive evaluation help?

Evaluate an n-coefficient polynomial at n values
$\Rightarrow$ Evaluate two $\frac{n}{2}$-coefficient polynomials at $\frac{n}{2}$ values

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$T(n) = O(n \log n)$$

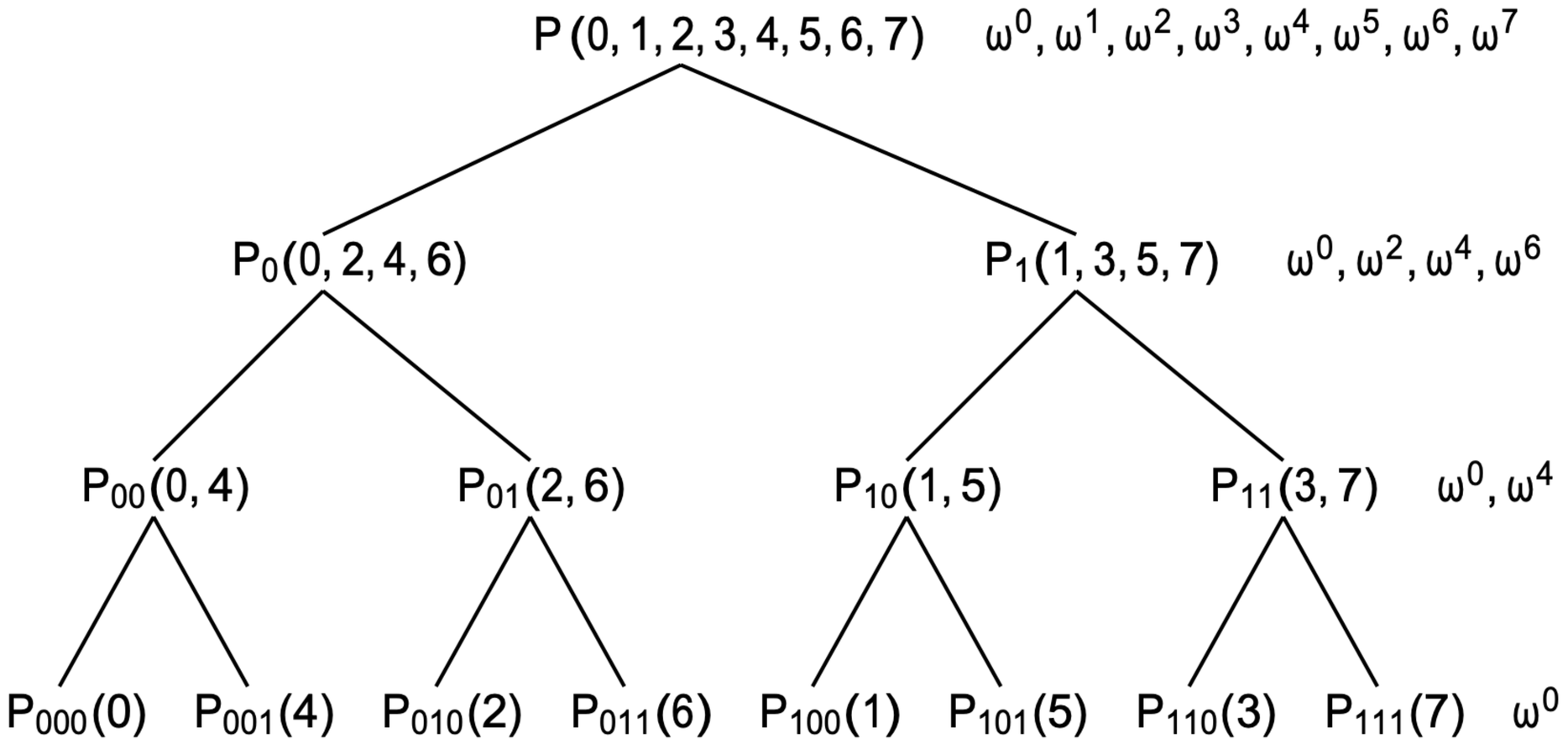
Representing Recursive Polynomials

Example:

- $P(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$
  Represent as $P(0, 1, 2, 3, 4, 5, 6, 7)$
- $P_e(x) = a_0 + a_2x + a_4x^2 + a_6x^3$
  $P_o(x) = a_1 + a_3x + a_5x^2 + a_7x^3$
  Represent as $P_0(0, 2, 4, 6)$ and $P_1(1, 3, 5, 7)$
- Similarly, $P_{00}(0, 4)$, $P_{01}(2, 6)$, $P_{10}(1, 5)$, $P_{11}(3, 7)$, etc.

- $P(\omega) = P_0(\omega^2) + \omega P_1(\omega^2)$
- $P(\omega^5) = P(-\omega) = P_0(\omega^2) - \omega P_1(\omega^2) = P_0(\omega^2) + \omega^5 P_1(\omega^2)$
- $P(\omega^3) = P_0(\omega^6) + \omega^3 P_1(\omega^6)$
- $P(\omega^7) = P(-\omega^3) = P_0(\omega^6) - \omega^3 P_1(\omega^6) = P_0(\omega^6) + \omega^7 P_1(\omega^6)$
- $P_0(\omega^2) = P_{00}(\omega^4) + \omega^2 P_{01}(\omega^4)$
- $P_0(\omega^6) = P_0(-\omega^2) = P_{00}(\omega^4) - \omega^2 P_{01}(\omega^4) =$
  $P_{00}(\omega^4) + \omega^6 P_{01}(\omega^4)$

Recursive Polynomial Evaluation



Polynomial evaluation with matrix-vector product

$$\begin{bmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

$$\begin{bmatrix} P(\omega^0) \\ P(\omega^1) \\ P(\omega^2) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^1 & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

$$P(\vec{\omega}) = M(\omega)\vec{a}$$

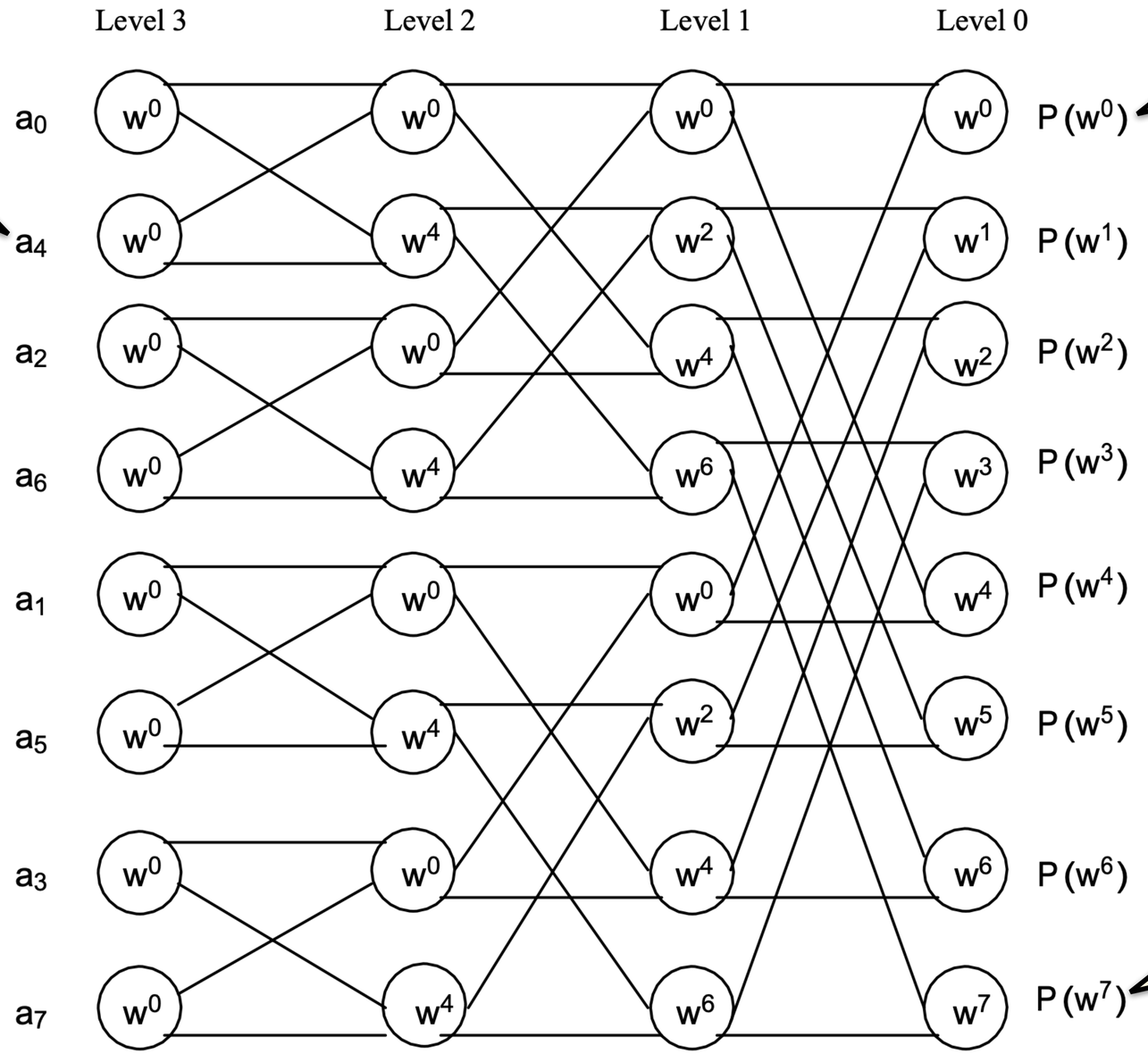
DFT: Discrete Fourier Transform – Multiplying given $n \times 1$ vector $\vec{a}$ with the special matrix $M(\omega)$

FFT: Fast Fourier Transform – The polynomial based algorithm to compute the matrix vector product in $O(n \log n)$ time

Parallel FFT Algorithm

Start with coefficient reversing bit string of rank

Rank i computes evaluation at ω^i



Horizontal connections are left shifting $\lg p$-bit rank index

Parallel FFT Algorithm

Reverse bit string - get that coefficient

$u \leftarrow a_{rev(rank)}$

for $i = 0$ to $d - 1$ do

- Send u to and receive v from $rank \oplus 2^i$.
- $m = \omega^{rank << (d-i-1)}$
- If i^{th} bit of $rank$ is zero

  $u = u + mv$

  else

  $u = v + mu$

endfor

both are
left child + m * right child

Solving Larger Problems

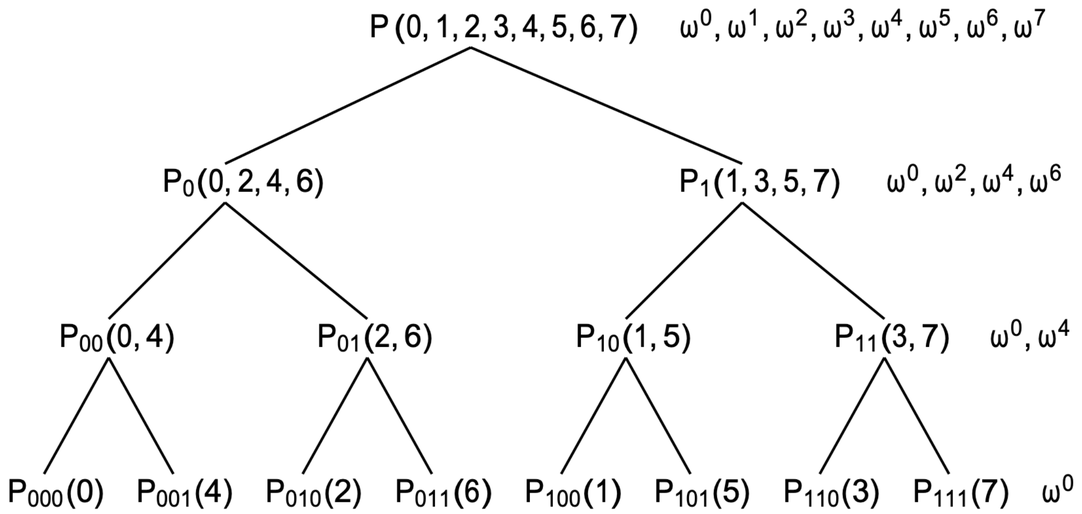
Consider DFT of size n on p processors.

Map the divide and conquer tree to processors.

The p subtrees at level $\log p$ are assigned to the p processors.

Top $\log p$ levels are computed using all the p processors.

- Computation Time $= O\left(\frac{n \log n}{p}\right)$
- Communication Time $= O\left(\tau \log p + \mu \frac{n}{p} \log p\right)$



What about Step 3?

Compute coefficients of a polynomial, given its value at $\omega^0, \omega^1, \omega^2, \ldots$

$$P(\vec{\omega}) = M(\omega)\vec{a}$$

If we are given $P(\vec{\omega})$ and need to find $\vec{a}$,

$$\vec{a} = [M(\omega)]^{-1} P(\vec{\omega})$$

normally $O(n^3)$ to invert the matrix

could precompute inverses, but depends on n

Lemma

Lemma: $[M(\omega)]^{-1} = \frac{1}{n} M\left(\frac{1}{\omega}\right)$

Proof: Let m_{ij} and m'_{ij} denote the ij^{th} entry of $M(\omega)$ and $M\left(\frac{1}{\omega}\right)$, respectively. Let $R = M(\omega) \times M\left(\frac{1}{\omega}\right)$.

$$r_{ij} = \sum_{k=0}^{n-1} m_{ik} m'_{kj} = \sum_{k=0}^{n-1} \omega^{ik} \left(\frac{1}{\omega}\right)^{kj} = \sum_{k=0}^{n-1} \omega^{k(i-j)}$$

Case I: $i = j$

$$\sum_{k=0}^{n-1} \omega^{k(i-j)} = \sum_{k=0}^{n-1} \omega^{0} = n$$

diagonals all n

Case II: $i \neq j$

$$\sum_{k=0}^{n-1} \omega^{k(i-j)} = \omega^{0} + \omega^{i-j} + \omega^{2(i-j)} + \ldots + \omega^{(n-1)(i-j)}$$

geometric

$$\frac{\left(\omega^{n(i-j)} - 1\right)}{\omega^{(i-j)} - 1} = 0$$

numerator always 1

Step 3 Strategy

$$\vec{a} = [M(\omega)]^{-1} P(\vec{\omega}) = \frac{1}{n} M\left(\frac{1}{\omega}\right) P(\vec{\omega})$$

ω is a primitive n^{th} root of $1 \Rightarrow \frac{1}{\omega}$ is also.

$M\left(\frac{1}{\omega}\right) P(\vec{\omega})$ can be computed using FFT!

Computing the coefficients of a polynomial whose values at the powers of ω are given is equivalent to evaluating a polynomial, whose coefficients are given by these values, at powers of $\frac{1}{\omega}$!