Lecture 35 The Fast Fourier Transform (FFT)

Consider two polynomials

$$\begin{aligned} f(x) &= a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \\ g(x) &= b_0 + b_1 x + b_2 x^2 + \ldots + b_m x^m \ . \end{aligned}$$

We can represent these two polynomials as vectors of some length $N \geq n + m + 1$. The i^{th} element of the vector is the coefficient of x^i.

$$\begin{aligned} f &= (a_0, a_1, a_2, \ldots, a_n, 0, 0, \ldots, 0) \\ g &= (b_0, b_1, b_2, \ldots, b_m, 0, 0, \ldots, 0) \ . \end{aligned} \tag{53}$$

The product of f and g will then be represented by the vector

$$(a_0 b_0, a_1 b_0 + a_0 b_1, a_2 b_0 + a_1 b_1 + a_0 b_2, \ldots) \ .$$

This vector is called the *convolution* of the vectors (53).

The obvious way to compute the convolution of two vectors takes N^2 processors and $\log N$ time. We would like to reduce the processor bound to N. To do this, we will use a different representation of polynomials. Recall that a polynomial of degree $N-1$ is uniquely determined by its values on N data points. Thus if we have N distinct data points $\xi_0, \xi_1, \ldots, \xi_{N-1}$, we can represent the polynomial f by the vector

$$(f(\xi_0), f(\xi_1), f(\xi_2), \ldots, f(\xi_{N-1})) \ . \tag{54}$$

The nice thing about this representation is that since

$$fg(\xi_i) \quad = \quad f(\xi_i)g(\xi_i) \ ,$$

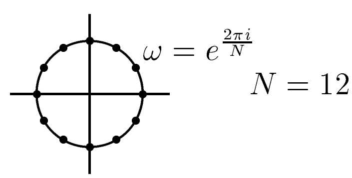
we can calculate the product of two polynomials by doing a componentwise product of the two vectors in constant time with N processors, provided the degree of the product is at most $N-1$.

The problem now is to find a way to convert from one representation to the other. For any choice of ξ_i, we can convert from (53) to (54) by evaluating the polynomials on the ξ_i; this amounts to multiplying (53) by the matrix

$$\begin{bmatrix} 1 & \xi_0 & \xi_0^2 & \cdots & \xi_0^{N-1} \\ 1 & \xi_1 & \xi_1^2 & \cdots & \xi_1^{N-1} \\ 1 & \xi_2 & \xi_2^2 & \cdots & \xi_2^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \xi_{N-1} & \xi_{N-1}^2 & \cdots & \xi_{N-1}^{N-1} \end{bmatrix} \tag{55}$$

called a *Vandermonde matrix.* We can convert back by interpolation, which amounts to multiplying (54) by the inverse of the matrix (55).

Judicious choice of the ξ_i can make this conversion very efficient. If we are working in a field containing N^{th} roots of unity (roots of the polynomial x^N-1) and a multiplicative inverse of N (*i.e.*, the characteristic of the field does not divide N), then we can get very efficient conversion algorithms by taking the ξ_i to be the N^{th} roots of unity. For example, in the complex numbers $\mathcal{C}$, let $\omega = e^{\frac{2\pi i}{N}}$ and take $\xi_i = \omega^i$. These points lie uniformly spaced on the complex unit circle (recall that to multiply two complex numbers, you add their angles and multiply their lengths).



The N^{th} roots of unity form a cyclic group under multiplication. An N^{th} root of unity ξ is called *primitive* ([3] uses the term *principal*) if it is a generator of this group, *i.e.* if every N^{th} root of unity is some power of ξ. Not all N^{th} roots of unity are primitive; for $N = 12$ in $\mathcal{C}$, the primitive roots are ω, ω^5, ω^7, and ω^{11}. The root ω^2 is not primitive, because its powers are all of the form ω^{2k}, so it is impossible to obtain odd powers of ω. In general, if ξ is a primitive root, then ξ^k is a primitive root if and only if k and N are relatively prime.

Over any field containing all N^{th} roots of unity, the polynomial $x^N - 1$ factors into linear factors

$$x^N - 1 \quad = \quad \prod_{i=0}^{N-1} (x - \omega^i) \ ,$$

where ω is a primitive N^{th} root of unity. This is because each of the N^{th} roots of unity is a root of $x^N - 1$, and there can be at most N of them. Since

$$x^N - 1 \quad = \quad (x-1)(x^{N-1} + x^{N-2} + \cdots + x + 1) \ ,$$

every N^{th} root of unity except $\omega^0 = 1$ is a root of the polynomial

$$\sum_{j=0}^{N-1} x^j \ .$$

This gives the following technical property, which we will find useful:

$$\sum_{j=0}^{N-1} w^{ij} \quad = \quad \begin{cases} 0 \ , & \text{if } i \not\equiv 0 \bmod N \\ N \ , & \text{otherwise.} \end{cases} \tag{56}$$

The $N \times N$ Vandermonde matrix (55) for these data points has as its ij^{th} element ω^{ij}, $0 \leq i, j \leq N-1$. We denote this matrix F_N. When applied to a vector containing the coefficients of a polynomial

$$f(x) \quad = \quad a_0 + a_1 x + \cdots + a_{N-1}x^{N-1} \ ,$$

F_N gives the vector of values of f at the N roots of unity.

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2N-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2N-2} & \cdots & \omega^{(N-1)^2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{N-1} \end{bmatrix} = \begin{bmatrix} f(1) \\ f(\omega) \\ f(\omega^2) \\ \vdots \\ f(\omega^{N-1}) \end{bmatrix}$$

The linear map represented by the matrix F_N is called the *discrete Fourier transform.*

The inverse of F_N is particularly easy to describe: its ij^{th} element is

$$(F_N^{-1})_{ij} \quad = \quad \frac{\omega^{-ij}}{N} \ .$$

Thus F_N^{-1} is $\frac{1}{N}$ times the Fourier transform matrix of a different primitive N^{th} root of unity, namely $\omega^{-1} = \omega^{N-1}$. To show that F_N and F_N^{-1} are indeed inverses, we just calculate their product, using property (56) at the critical step:

$$\begin{aligned} (F_N \cdot F_N^{-1})_{ij} \quad &= \quad \sum_{k=0}^{N-1} \omega^{ik} \cdot \frac{\omega^{-kj}}{N} \\ &= \quad \frac{1}{N} \sum_{k=0}^{N-1} \omega^{k(i-j)} \\ &= \quad \begin{cases} 1 \ , & \text{if } i = j \\ 0 \ , & \text{otherwise,} \end{cases} \end{aligned}$$

thus $F_N F_N^{-1}$ is the identity matrix.

Now we want to find a way to compute $F_N f$ quickly, where

$$f = (a_0, a_1, \ldots, a_{N-1})$$

is the vector of coefficients of the polynomial $f(x)$. We use a divide-and-conquer approach in which we split f into two polynomials each of size $\frac{N}{2}$ (assume for simplicity that N is a power of 2), apply $F_{\frac{N}{2}}$ to each of them in parallel, then combine the two results to form $F_N f$.

Given

$$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{N-1} x^{N-1} ,$$

define

$$\begin{aligned} f_0(x) &= a_0 + a_2 x^2 + a_4 x^4 + \ldots + a_{N-2} x^{N-2} \\ \widehat{f}_0(x) &= a_0 + a_2 x + a_4 x^2 + \ldots + a_{N-2} x^{\frac{N}{2}-1} \\ f_1(x) &= a_1 + a_3 x^2 + a_5 x^4 + \ldots + a_{N-1} x^{N-2} \\ \widehat{f}_1(x) &= a_1 + a_3 x + a_5 x^2 + \ldots + a_{N-1} x^{\frac{N}{2}-1} . \end{aligned}$$

Then

$$\begin{aligned} f(x) &= f_0(x) + x f_1(x) \\ f_0(x) &= \widehat{f}_0(x) \circ x^2 \\ f_1(x) &= \widehat{f}_1(x) \circ x^2 \end{aligned}$$

where $\circ$ represents functional composition (substitute the right polynomial for the variable in the left polynomial). Both $\widehat{f}_0$ and $\widehat{f}_1$ have degree at most $\frac{N}{2} - 1$. We recursively apply $F_{\frac{N}{2}}$ to the vectors $\widehat{f}_0 = (a_0, a_2, \ldots, a_{N-2})$ and $\widehat{f}_1 = (a_1, a_3, \ldots, a_{N-1})$ to get $F_{\frac{N}{2}} f_0$ and $F_{\frac{N}{2}} f_1$. The primitive $\frac{N}{2}^{\text{th}}$ root of unity used in the formation of $F_{\frac{N}{2}}$ is ω^2.

Now we show that the N-vector $F_N f_0$ is obtained by concatenating two copies of the $\frac{N}{2}$-vector $F_{\frac{N}{2}} \widehat{f}_0$, and similarly for f_1. The i^{th} element of $F_N f_0$ is

$$\begin{aligned} f_0(\omega^i) &= (\widehat{f}_0 \circ x^2)(\omega^i) \\ &= \widehat{f}_0(\omega^{2i}) , \end{aligned}$$

which is the i^{th} mod $\frac{N}{2}$ element of $F_{\frac{N}{2}} \widehat{f}_0$. The argument is similar for f_1.

Finally

$$\begin{aligned} F_N f &= F_N(f_0 + x f_1) \\ &= F_N f_0 + F_N(x f_1) \\ &= F_N f_0 + F_N x \cdot F_N f_1 , \end{aligned}$$

where $\cdot$ represents componentwise multiplication. We have already computed $F_N f_0$ and $F_N f_1$ by recursively computing the Fourier transform of two vectors of size $\frac{N}{2}$; and

$$F_N x \quad = \quad (1, \omega, \omega^2, \ldots, \omega^{N-1}) \ ,$$

so we have all we need to compute $F_N f$.

With N processors, it takes us constant time to split f into $\widehat{f}_0$ and $\widehat{f}_1$. We then do two recursive calls in parallel to calculate $F_N f_0$ and $F_N f_1$, each using $\frac{N}{2}$ processors. Finally, it takes constant time to recombine the results to get $F_N f$. Therefore, the algorithm uses $O(\log N)$ time and N processors.

This gives a very efficient parallel algorithm for multiplying two polynomials: compute their Fourier transforms, multiply the resulting vectors componentwise, then take the inverse Fourier transform. The entire algorithm takes $O(\log N)$ time and N processors.

It is interesting to ask what happens when the degrees of the polynomials are so large that the degree of their product exceeds $N-1$. The answer is that terms that fall off the right side of the vector wrap around; in other words, the coefficient of the term x^{N+i} in the product is added to the coefficient of x^i. Mathematically, what is going on is that the product of the two polynomials is being computed modulo the polynomial $x^N - 1$:

$$F_N^{-1}(F_N f \cdot F_N g) \quad = \quad fg \bmod x^N - 1 \ .$$

A fancy way of saying this is that the Fourier transform gives an isomorphism

$$F_N : k[x]/(x^N - 1) \quad \rightarrow \quad k^N$$

between two N-dimensional algebras over the field k, namely the algebra of polynomials mod $x^N - 1$ with ordinary polynomial multiplication and the direct product k^N with componentwise multiplication.

The parallel algorithm for the FFT given here is essentially implicit in the 1965 paper of Cooley and Tukey [24], although that was well before anyone had ever heard of NC.