2-Radix Fast Fourier Transform for Polynomial Multiplication

Nur Alam Hasabie 13517096
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha No.10 Bandung 40132, Indonesia
alamhasabie165@gmail.com

***Abstract*— Fourier Transform is one of the most important transformation in engineering. However, beside engineering and signal processing, Fourier Transform also provides an alternative divide-and-conquer approach to polynomial multiplication. The solution would be based on the common 2-Radix Fast Fourier Transform algorithm and its inverse transform. A comparison with the more straightforward approach will also be discussed, along with brief comparison with other polynomial multiplication algorithms known to date.**

Keywords— Fast Fourier Transform, Polynomial Multiplication, Divide and Conquer

I. Introduction

Fourier transform is one of the most fundamental tools in wave analysis. Given a wave with certain equation, one could decompose the wave into its sinusoidal components. The idea lies in the fact that every periodic function can be exactly approached by sum of Fourier Series.

However, its applications are not limited to the fields stated previously. One interesting application found regarding Fourier Transform is polynomial multiplication. Previous algorithms, such as Karatsuba algorithm, provided a solution with time complexity $O(n^{log3})$, an improvement over the straightforward approach with $O(n^2)$ time complexity. However, Fast Fourier Transform (FFT) algorithm can provide a faster *O(nlogn)* working algorithm for polynomial multiplication. There are properties of FFT which can be exploited by evaluating the polynomials with the root of unity. The root of unity is periodic, therefore there are some values which not needed to be computed twice during the evaluation. The paper , thus ,will discuss the steps and several proofs necessary to accomplish a better polynomial multiplication algorithm with FFT.

II. Theories

A. Discrete Fourier Transform

Discrete fourier transform is the discrete equivalent of continuous Fourier Transform (as in that there are finite data separated by time interval T_i).

The continuous Fourier Transform is defined by :

$$F(j\omega) = \int_{-\infty}^{\infty} f(t)\, e^{-j\omega t} dt \qquad (1)$$

To transform the original equation into a discrete one (the type of transformation that will be primarily used in this paper), consider N samples *f [0] , f [1] , f [2] ... , f [N-1]* taken from the source *f (t)* . Every sample can be considered as an impulse with area *f [k] = f(k)T*. *T* corresponds to the interval, therefore *dt* can be approached with T. The integrands would only be defined in the sample range, that is[1] :

$$F(j\omega) = \int_{0}^{(N-1)T} f(t)\, e^{-j\omega t} dt \qquad (2)$$

Therefore,

$$F(j\omega) = \sum_{k=0}^{N-1} f[k]\, e^{-j\omega kT} \qquad (3)$$

Treating the data as periodic (ie. *f[0]* to *f [N-1]* is same as *f[N]* to *f[2N-1]*), then DFT can be evaluated at its fundamental frequency (*1/(NT)* Hz). Thus [2],

$$F[n] = \sum_{k=0}^{N-1} f[k] e^{-2\pi nkj/N} \qquad (4)$$

It is more common to write $\omega = e^{2\pi j/N}$.

Eq. 4 defined as the DFT. However, it is sometimes easier to represent such linear equations with matrix.

DFT is represented by a matrix :

$$\begin{bmatrix} F[0] \\ F[1] \\ F[0] \\ .. \\ F[N-1] \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & ... & 1 \\ 1 & W^1 & W^2 & & W^{N-1} \\ 1 & W^2 & W^4 & & W^{N-2} \\ ... & & & & \\ 1 & W^{N-1} & W^{N-2} & ... & W \end{bmatrix} \begin{bmatrix} f[0] \\ f[1] \\ f[2] \\ ... \\ f[N-1] \end{bmatrix}$$

(5)

Fig 1. Matrix representation of Fourier Transform

W is defined as $exp(-2\pi i/N)$

Also, stated without proof, the inverse transform,

$$f[n] = \frac{1}{N} \sum_{k=0}^{N-1} f[k] e^{-2\pi nkj/N} \qquad (6)$$

The inverse would play a big role in the Fast Fourier Transform later.

Discrete Fourier Transform found many fields of applications, especially in signal processing. Computers, obviously, do not work with continuous data. Computers work with samples taken from the analyzed waves. The continuous transformation therefore can be approached with the discrete one.

B. Divide and Conquer

Divide and Conquer is one of the most important approach in solving computation problems. This idea is introduced mainly because approaches to find better polynomial multiplication algorithms have utilized this idea. The main idea behind the approach is to divide the problem into several smaller instances of the same problem, and combine the result of those smaller problems to obtain the original problem (the "conquer" part).

The basic algorithm is as the following :

1. Divide the problems into smaller instances of the same problem
2. For each smaller problems, obtain the answer. This part of the algorithm is where the recurrences are.
3. Combine all the answers to obtain the general solution of the problem

As for each recurrence, the base should be defined. The base for divide and conquer approach is not strictly defined, as long as it is beneficial for the sake of performance. One could define a base case of certain size, in which it could be solved fast with more naïve algorithms.

Divide and algorithm approach has been applied to find better solutions of computational problems when compared to "brute force" approach, although not necessarily as such for several problems.

To determine the time complexity of a divide and conquer solution, Master theorem is used[3]. Master theorem provides solution in Big-O notation for recurrence relations, notable feature in divide and conquer algorithms. The recurrences considered have the form as the following :

$$T(n) = aT(n/b) + f(n) \qquad (7)$$

That is, the problem is divided into a smaller problems with n/b size , and additional work of $f(n)$ (usually for combining). There are three cases for $T(n)$, which are :

1. If it is true that $f(n) = \Theta(n^c)$ where $c < log_b a$, then :
$$T(n) = \Theta(n^{log_b a}) \qquad (8)$$
2. If it is true for some constant $k \geq 0$ that $f(n) = \Theta(n^c log^k n)$ where $c = log_b a$, then
$$T(n) = \Theta(n^c log^{k+1} n) \qquad (9)$$
3. If it is true that $f(n) = \Theta(n^c)$ where $c > log_b a$, then :
$$T(n) = \Theta(f(n)) \qquad (10)$$

C. 2-Radix Fast Fourier Transform

2-Radix Fast Fourier Transform (FFT) is a part of a family of Fast Fourier Transform. It is the most common implementation of Cooley-Tukey algorithm , and can be explained solely from the perspective of Fourier Transform.

The idea behind the algorithm is exploiting the symmetry within the definition of discrete Fourier Transform. Consider the following discrete Fourier Transform :

$$F[n] = \sum_{k=0}^{N-1} f[k] e^{-j2\pi nk/N} \qquad (11)$$

To find the symmetry, $F[n]$ can be written as :

$$F[n] = \sum_{m=0}^{N/2-1} f[2m] e^{-j2\pi n(2m)/N} + \sum_{m=0}^{N/2-1} f[2m+1] e^{-j2\pi n(2m+1)/N} \qquad (12)$$

The first summation is the summation of all terms with even indices, and the second summation is the summation of all terms with odd indices. The equation can be further arranged :

$$F[n] = \sum_{m=0}^{N/2-1} f[2m] e^{-j2\pi nm/(N/2)} + e^{-2\pi jn/N} \sum_{m=0}^{N/2-1} f[2m+1] e^{-j2\pi nm/(N/2)} \qquad (13)$$

Notices that the first summation is the discrete Fourier Transformation for even terms, and the second summation is the discrete Fourier Transform for odd terms.

$$E[n] = \sum_{m=0}^{N_0} f[2m] e^{-2\pi j mn/(N_0+1)},\ N_0 = N/2 - 1 \qquad (14)$$

$$O[n] = \sum_{m=0}^{N_0} f[2m+1] e^{-2\pi j mn/(N_0+1)},\ N_0 = N/2 - 1 \qquad (15)$$

Therefore :

$$F[n] = E[n] + e^{-2\pi j n/N}\, O[n] \qquad (16)$$

Also, consider the case of $n + N/2$:

$$F[n + N/2] = E[n + N/2] + e^{-2\pi j (n+N/2)/N} O[n + N/2] \quad (17)$$

$$E[n + N/2] = \sum_{m=0}^{N_0} f[2m] e^{-2\pi j m(n+N_0+1)/(N_0+1)},\ N_0 = N/2 - 1 \quad (18)$$

$$E[n + N/2] = \sum_{m=0}^{N_0} f[2m] e^{-2\pi j mn/(N_0+1)} e^{-2\pi j m} \qquad (19)$$

Due that $e^{i\Theta} = cos(\theta) + i\ sin\ (\theta)$, it follows that $e^{-2\pi j m} = 1$. Therefore $E[n + N/2] = E[n]$.

Also, without proof, it is easily seen that $O[n] = O[n + N/2]$. However , $e^{-2\pi j (n+N/2)/N} = -e^{-2\pi j (n))/N}$, therefore :

$$F[n + N/2] = E[n] - e^{-2\pi j n/N} O[n],$$

$$F[n] = E[n] + e^{-2\pi j n/N}\, O[n] \qquad (20)$$

It can be deduced easily that by finding E_n and O_n , all values can be determined with less computation , compared to straightforwardly compute the summation for each $F\ [k]$.

To find the complexity, recall the recurrence form of $T(n) = aT(n/b) + f(n)$. radix-2 FFT algorithm divides the problem into 2 smaller sub-problems, each is $N/2$ in size. The time do the summation will be $f(n) = \Theta\ (n^1)$. Since c = 1 = $log_b a$ = $log_2 2$, therefore the complexity of radix-2 FFT algorithm will be :

$$T(n) = \Theta\ (nlog^{0+1} n) = \Theta\ (nlogn) \qquad (21)$$

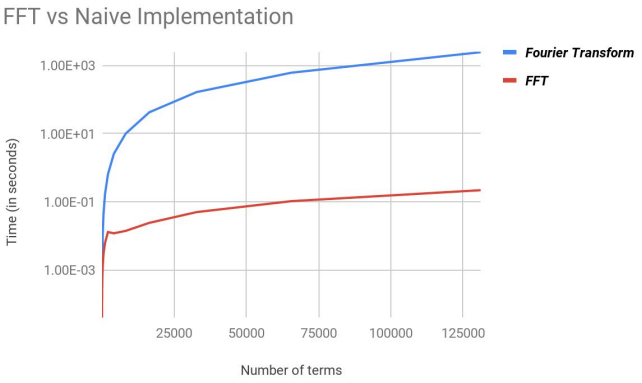

Fig. 2. Time complexity comparison between the straightforward algorithm and FFT algorithm. Notice the logarithmic scale of y-axis. Source : Author's document

However, as the paper titled, the algorithm discussed here will be limited to radix-2 FFT algorithm (the number of samples should be the power of 2) . A more general form for arbitrary N samples exists, but it is out of the scope of this paper.

D. Polynomial Multiplication and Fast Fourier Transform

In general , given two polynomials in coefficients representation [4] :

$$f(x) = \sum_{i=0}^{N-1} a_i x^i\ ,\ g(x) = \sum_{i=0}^{N-1} b_i x^i \qquad (22)$$

The multiplication of both polynomials :

$$f(x)g(x) = (\sum_{i=0}^{N-1} a_i x^i)(\sum_{i=0}^{N-1} b_i x^i) \qquad (23)$$

$$f(x)g(x) = (\sum_{i=0}^{N-1} \sum_{j=0}^{M-1} a_i b_j x^{i+j}) \qquad (24)$$

Or, in terms of some c :

$$f(x)g(x) = C(x) = \sum_{i=0}^{2N-1} c_i x^i,\ c_i = \sum_{j=0}^{i} a_i b_{i-j} \qquad (25)$$

The sequence of c can be viewed as *convolution* of polynomial $f(x)$ and $g(x)$. The straightforward implementation of the algorithm has $O(n^2)$ time complexity.

Another idea to be presented is the point-value representation of polynomials. Given a n degree-bound polynomial $A(x)$, its point-value representation is [5]:

$$\{(x_0, y_0), (x_1, y_1), (x_2, y_2), ..., (x_{n-1}, y_{n-1})\} \qquad (26)$$

Such that :

1. $\forall i \neq j,\ x_i \neq x_j$
2. $\forall k,\ y_k = A(x_k)$

The representation means that all pairs of point-value can determine an unique polynomial with degree-bound n. Moreover , Point-value representation is convenient to do certain operations, including addition and multiplication. If the convolution $C(x) = A(x)\ B(x)$ holds, then $\forall k, C(x_k) = A(x_k)\ B(x_k)$. This operation only has $O(n)$ time complexity, much faster than the original . Therefore it is hoped that the evaluation of $C(x)$ can be done in this representation. However, converting the coefficients representation to point-value representation will have $O(n^2)$ time complexity (using Horner's method to evaluate the values).

Even after $C(x)$ is determined in its point-value representation, it should be converted back to its coefficients representation. The most common fast algorithm for this, Lagrange's formula for the interpolation, has $O(n^2)$ time complexity.

Using this approach, the algorithm to find the convolution $C(x)$ has $O(n^2)$ time complexity for both the straightforward approach and the one through point-value representation.

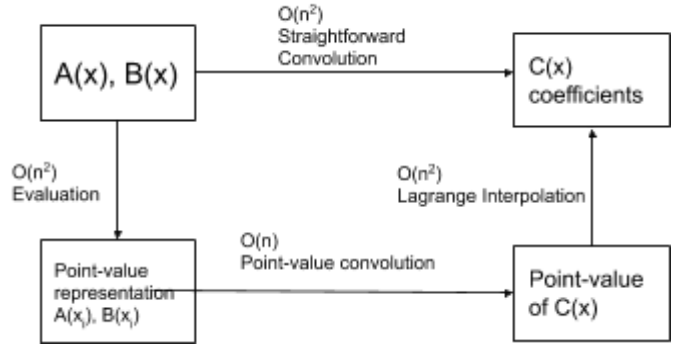

Fig 3. Representation of states in finding the convolution C(x) = A(x) B(x) . Notices that point-value representation does not optimize the algorithm

However, there is still flexibility in choosing how to evaluate the polynomial. First, consider the value $w_n = e^{2j\pi/n}$. (This value is called as root unity value). Evaluate this value to obtain the point-value presentation of $A(x)$:

$$y_k = A(\omega_n^k)$$

$$y_k = \sum_{p=0}^{n-1} a_n \omega_n^{kp} = \sum_{p=0}^{n-1} a_n \omega_n^{kp} = \sum_{p=0}^{n-1} a_n e^{2\pi kpj/N} \qquad (27)$$

Note that the form is equivalent to discrete Fourier Transform in Eq. 3. Therefore, the evaluation process is modified into FFT process, which has $O(nlogn)$ time complexity.

In other sense, the problem is transformed from evaluating $A(x)$ for $\omega_n^0, \omega_n^1, \omega_n^2, ...\omega_n^{n-1}$ into evaluating [6] :

$$A(x) = A^{even}(x) + xA^{odd}(x) \qquad [28]$$

The second part of the algorithm is the convolution with point-value representation. As the convolution would only have $O(n)$ time complexity, until this step the algorithm has $O(nlogn)$ time complexity. The third part , converting the point-value representation of the convolution to the coefficients representation would be replaced by applying inverse of the FFT. The inverse transformation can also be applied to the recursive FFT using the conjugates of the complex numbers.

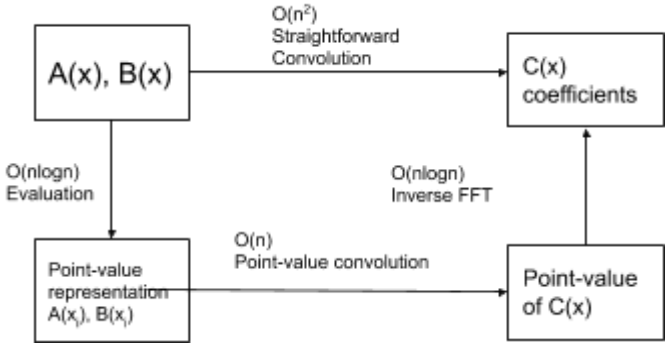

Fig 4. Representation of states in finding the convolution C(x) = A(x) B(x) .

III. Implementation Of fast fourier transform for polynomial multiplication

A. Implementation of 2-Radix Fast Fourier Transform

The pseudocode for Radix-2 FFT algorithm and its inverse is written in following illustrations :

```
getCoefficients :
   // n degree polynomials
   firstPolyCoeffs[2n-1]
   secondPolyCoeffs[2n-1]
   resultPolyCoeffs[2n-1]

    Fill elements from 0 to n-1 on
    firstPolyCoeffs and secondPolyCoeffs with
    polynomial coefficients

    Fill elements from n to 2n-1 on
    firstPolyCoeffst and secondPolyCoeffs
    with 0

    FFT2Radix(firstPolyCoeffs)
    FFT2Radix(secondPolyCoeffs)
    For each elements in each polynom :
    resultPolyCoeffs =
    firstPolyCoefss*secondPolyCoeffs

    Inverse(resultPolyCoeffs)
```

Fig 5. Pseudocode for 2-radix FFT. The pseudocode thus was translated in a small C++ program.

```
FFT2Radix(x):

  // x is array of complex
  N = length of x

  // Divide
  x_even = x[even numbers]
  x_odd = x[odd_numbers]

  //conquer
  FFT2Radix(x_even)
  FFT2Radix(x_odd)

  for k=0 to N/2 :
    Complex t = exp(-2πik/N)*x_odd[k]
    x[k] = x_even[k] + t
    x[k + N/2] = x_even[k] - t

InverseOfTransform(x) :

  // x is array of complex number
  x<- conjugation of all x

  FFT2Radix(x)

  x<- conjugation of all x
  x<- x divide by size of x
```

Fig 6. Pseudo Code for FFT multiplication

The program should be tested with several study cases. Two small test cases are given :

1. First, case of two 1-degree polynomials. The polynomials tested are $f(x) = 3 + x$ and $g(x) = 2 + x$. The program then prints then coefficients of $g(x) = f(x)\ g(x)$, both of its imaginary and real part. The expected output is (0,0,0) for its imaginary parts and (6,5,1) for its real part. The answer provided by program is (6,5,1,0) for the real part and (-0, 1.53081e-16 , - 0, -1.53081e-16) . There seems to be an inaccuracy in the imaginary part, with most probable cause is number round off by program

2. The second case will be two identical polynomials $(x+1)^4$. The expected answer is expected to be $(x+1)^8$, which can be verified easily by Pascal Triangle. Moreover, the imaginary part of all terms should be 0. However, the result printed by program is :

   $$Re = \{1,8,28,56,70,56,28,8,1\}$$

   $$Im = \{2.99e\text{-}15, -1.33e\text{-}15, -3.85e\text{-}32, -1.78e\text{-}15, -7.45e\text{-}15, -5.33e\text{-}15, -2.65e\text{-}15, 2.00e\text{-}15, 2.99e\text{-}15\}$$

   There seems to be an increase of error in the imaginary part of each terms, however the real part of the terms are verified to be correct.

It cannot be deduced, however, that the program would always produce correct answer, as 2 sample cases is insufficient to determine such claim. However, under the assumption that the program does give the correct result, the performance test can be conducted.

C. Comparing with Straightforward Implementation of Polynomial Multiplication

After implementing the algorithm, a comparison with the straightforward algorithm should be conducted briefly. The easiest way write such algorithm is by employing Eq. 24. As discussed previously, the algorithm theoretically would have $O(n^2)$ time complexity.

A small test is ran on a computer (specification : x86_64 computer with 4 GB RAM and Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz) to compare between the two algorithm. The result shows that FFT algorithm is indeed much faster for large polynomials (for polynomial with degree of thousands), but the straightforward approach seems to run faster for smaller instances of the problem. There is a limit where divide and conquer approach would be a little bit inefficient, hypothetically due to recurrent function calling.

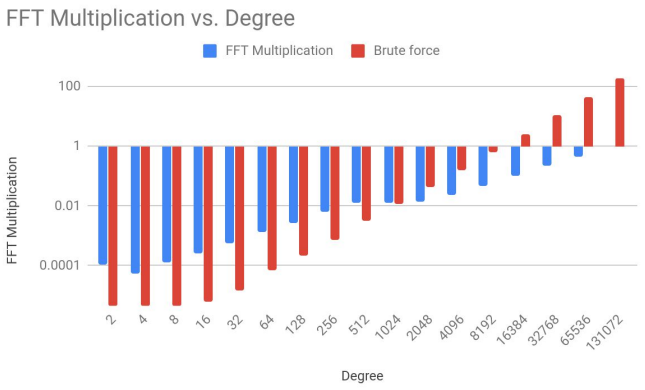

Fig 4. Comparison charts of execution-time between FFT multiplication and the more "brutish" algorithm. Note that FFT is still slower than brute-force until a certain size of instance. Source : Author's document

IV. Other Polynomial multiplication algorithms

This section provides other alternatives of polynomial multiplication algorithm, as to not limit the options to solve the problem.

Since only 2-radix FFT algorithm is discussed here, several generalizations should be introduced briefly. Some variations and generalizations including Cooley-Tukey algorithm for arbitrary size and split-radix FFT.

Another divide and conquer approach algorithm is the modification of Karatsuba's multiplication algorithm to be applied on polynomial. Weimerskirch and Paar [2006] has demonstrated a generalization of Karatsuba algorithm for arbitrary sized polynomials [7] .

Acknowledgment

Firstly, I would like to send my praise and my gratitude to the Gracious God, whose guidance and blessings and love has descended to the world , especially with Math. Mathematics, is indeed, the language and song of the universe.

Second, my gratitude to my parents, for keeping their love and support for me . I also want to express my gratitude to Mrs. Patria, my homeroom teacher in high school , for her steady support during my study.

I would also give my sincere gratitude to my beloved lecturer, Mr. Rinaldi. I've truly enjoyed the lectures, and give his students to explore things beyond what is taught.

Last but not least to the scientists and passionate people whose work I have referenced or seen. I've always intrigued by how complicated things are, yet some of you do explain it in a very flowing manner. I feel so satisfied after working on this paper.

References

[1] "Lecture 7 - The Discrete Fourier Transform" . No date or any other publishing information found. file available online at : http://www.robots.ox.ac.uk/~sjrob/Teaching/SP/l7.pdf

[2] S, Hagit, "The Fourier Transform - A Primer", Department of Computer Science, Brown University. November 1995. file available at : ftp://ftp.cs.brown.edu/pub/techreports/95/cs95-37.pdf

[3] Cormen, T.H., , Leiserson, C.E., Rivest, R.L, and C. Stein . Introduction to Algorithms, Third Edition . MIT Press. 2009. p94-97.

[4] Cormen, p. 898.

[5] Cormen, p. 901

[6] Cormen, p. 910

[7] W, Andre and P, Christof. "Generalizations of the Karatsuba Algorithm for Efficient Implementations". Cryptology ePrint Archive, Report 2006/224. 2006. file : https://eprint.iacr.org/2006/224.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 29 April 2012

Nur Alam Hasabie 13517096