INDIAN INSTITUTE OF TECHNOLOGY

ROORKEE

Autumn Session 2014

Project Report on

Implementation of FFT using Divide and Conquer Programming Approach

Submitted to-
Dr Durga Toshniwal
Associate Professor (CSE)

Submitted by-
Amala Thampi (14535003)
Nikita Jain (14535031)
Nishtha Behal (14535032)
Preeti Bansal (14535035)
Yashika Jain (14535054)

Contributions

Amala Thampi	:	Contributed in complexity analysis and in writing the report.
Nikita Jain	:	Contributed in complexity analysis and in writing the report.
Nishtha Behal	:	Contributed in the coding part.
Preeti Bansal	:	Contributed in the coding part.
Yashika Jain	:	Contributed in designing pseudo-code and in writing the report.

Table of Contents

Introduction

The Fast Fourier Transform (FFT) is an efficient algorithm to compute the Discrete Fourier Transform (DFT). There are many different FFT algorithms involving a wide range of mathematics, from simple complex-number arithmetic to group theory and number theory. FFT computes DFT of N points in $\Theta(N\log N)$ operations as compared to $\Theta(N^2)$ operations performed in conventional method.

Theory

The Discrete Fourier Transform-

In DFT, we have to calculate following polynomial-

$$A(x)=\sum_{j=0}^{n-1} a_j x^j$$

of degree-bound n at $\omega_n^0, \omega_n^1, \omega_n^2\ \omega_n^3, \ldots, \omega_n^{n-1}$

where $\omega_n = e^{2\pi i/n}$ is the **principal n^{th} root of unity.**

Here, A is given in coefficient form as $a = (a_0, a_1, a_2, \ldots, a_{n-1})$.
The result y_k, for k=0,1,...,n-1, by

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

$$= \sum_{j=0}^{n-1} a_j \quad \omega_n^{kj}$$

The vector $y = (y_0, y_1, \ldots, y_{n-1})$ is the **Discrete Fourier Transform(DFT)** of the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1})$.

The Fast Fourier Transform-

The FFT take advantage of the special property of the complex root of unity called ***twiddle factor.*** It takes can compute DFT of N points in $\Theta(N\log N)$.
The FFT method employs a divide-and-conquer strategy, using the even-indexed and odd-indexed coefficients of $A(x)$ separately to define the two new polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$ of degree-bound $n/2$:

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \ldots + a_{n-2}x^{n/2-1}$$
$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \ldots + a_{n-1}x^{n/2-1}$$

where, $A^{[0]}$ holds all the even-indexed coefficients of A and $A^{[1]}$ holds all the odd-indexed coefficients of A.
Therefore,

$$A(x) = A^{[0]}(x) + A^{[1]}(x)$$

So, instead of evaluating A(x) at $\omega_n^0, \omega_n^1, \omega_n^2\ \omega_n^3, \ldots, \omega_n^{n-1}$, we can evaluate $A^{[0]}(x)$ and $A^{[1]}(x)$ at $(\omega_n^0)^2, (\omega_n^1)^2, (\omega_n^2)^2, \ldots, (\omega_n^{n-1})^2$ and then combine the results using above equation.

According to the halving lemma, if $n>0$ is even, then the squares of the n complex n^{th} roots of unity are the $n/2$ complex $(n/2)^{th}$ roots of unity. Following this we can say that the points at which $A^{[0]}(x)$ and $A^{[1]}(x)$ are calculated, does not consists of n distinct values but only of $n/2$ complex $(n/2)^{th}$ roots of unity, with each root occurring exactly twice. Thus, we recursively solve these sub-problems which are having same structure at that of original problem, but the size is half of the original one. The implemented recursive FFT algorithm computes the DFT of an n-element vector $a = (a_0, a_1, \ldots, a_{n-1})$, where n is a power of 2.

For $y_0, y_1, \ldots, y_{n/2-1}$ we use:

$$y_k = y_k^{[0]} + \omega_n^k y_k^{[1]}$$
$$= A(\omega_n^k)$$

For $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$ we use:

$$y_{k+(n/2)} = y_k^{[0]} - \omega_n^k y_k^{[1]}$$
$$= A(\omega_n^{k+(n/2)})$$

Here we use the property that $\omega_n^{k+(n/2)} = -\omega_n^k$

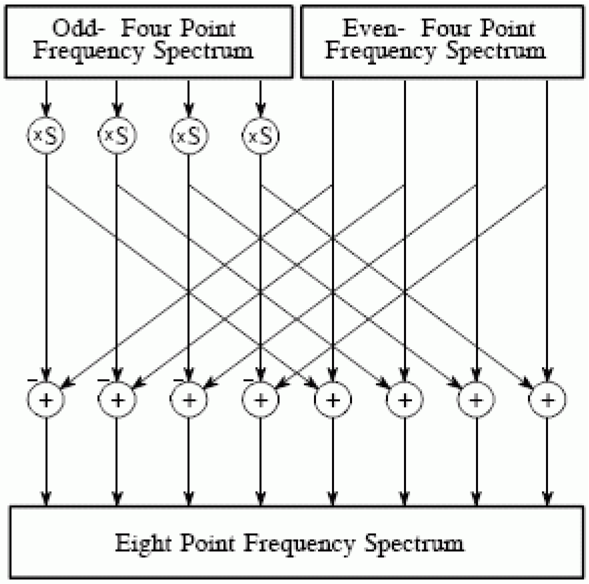

Fig. 1 Method of combining two 4-point frequency spectra into a single 8 point frequency spectrum. The xS operation means that the signal is multiplied by a sinusoid with an appropriately selected frequency.

Applications-

FFT introduces a substantial improvement, which made many DFT-based algorithms practical. FFTs can be used in wide variety of applications, like digital signal processing, solving partial differential equations and for quick multiplication of large numbers.

Complexity Analysis

1. **Brute Force**-

   In the brute force method we calculate each of the *n* components of the vector y in sequential manner, term by term.

   Total number of such terms = *n*

   Computing each term involves addition of *n* products. So the running time of this straightforward method would be $\Theta(n^2)$.

2. **Divide-and-Conquer-**

   In divide and conquer programming approach, invocation to each recursive call takes $\Theta(n)$ time, where *n* is length of the input vector. Thus, the recurrence relation becomes

   $$T(n) = 2T(n/2) + \Theta(n)$$
   $$= \Theta(n \log n)$$

3. **Dynamic Programming-**

   The dynamic programming approach is applicable to problems having properties of optimal substructure and overlapping sub-problems. But here the problem cannot be divided in such a manner so as to have overlapping sub-problems. So, we must not apply dynamic programming approach in order to solve the above problem.

4. **Greedy Algorithm-**

   Greedy algorithms are based on the property that a globally optimal solution can be derived by making locally optimal choices. This problem doesn't relate to this property as we have to compute each value and that too precisely. A nearly correct solution will not perform.

Pseudo-code

Input: An array of coefficients *coeff* and its length *lent.*

```
CAL_MULT(*a, *b)
t->real  := (a->real*b->real) - (a->imag*b->imag)
t->imag := (a->real*b->imag) + (a->imag*b->real)
return t

FFT(coeff, lent)
if(lent = 1)
        base->real := coeff[0]
        base->next := NULL
        return base

coeff_o := lent/2
coeff_e := lent-lent/2
e = o :=0
w->real := 1
wn->real := cos(6.28/lent)
wn->imag := sin(6.28/lent)

for i:=0 to lent-1
        if(i%2 = 0)
                coeff_e[e++] := coeff[i]
        else
                coeff_o[o++] := coeff[i]

y1 := fft(coeff_e, e)
y2 := fft(coeff_o, o)
k := lent/2

while(y1 is not empty)
        if(y = NULL and mid = NULL)
                y := start1
                mid := start2
        else
                y->next := r
                y := y->next
                mid := mid->next

        t := cal_mult(w,y2)
        y->real := y1->real + t->real
        y->imag := y1->imag + t->imag
        mid->real := (y1->real) - (t->real)
        mid->imag := (y1->imag) - (t->imag)
        w := cal_mult(w, wn)
        y1 := y1->next
        y2 := y2->next
y->next := start2
return start1
```

Code Complexity Analysis-

The above implementation is done using divide and conquer method. The time complexity analysis of FFT(coeff, lent) is as follows-

1. The FFT() function is called twice with input size of lent/2 resulting in 2T(n/2) (assuming lent is denoted by *n* in complexity analysis) as the size of input is divided into two halves every time as coeff_e (coefficient for even) and coeff_o (coefficient of odd) each of length n/2 in each recursion.

2. The while loop from line 19 to 34 in pseudo-code, takes Θ (n) complexity in each recursion.
   Hence the recurrence relation time complexity T(n) is

$$\mathbf{T(n) = 2T(n/2) + \Theta(n)}$$

   On solving the above recurrence relation using Master's theorem for divide and conquer we get complexity as Θ(nlogn).

C++ Code

```
#include<iostream>
#include<cmath>
#include <cstdlib>
using namespace std;

struct complex_num
{
   float real;
   float imag;
   complex_num *next;
}*start1,*start2;

complex_num* cal_mult(complex_num *a, complex_num *b)
{
   //function to multiply two complex numbers
   complex_num *t=(complex_num*)malloc(sizeof(complex_num));
   t->real=(a->real*b->real)-(a->imag*b->imag);
   t->imag=(a->real*b->imag)+(a->imag*b->real);
   return t;
}

complex_num* fft(int *coeff,int lent)
{
   complex_num *y1,*y2,*y=NULL,*mid=NULL,*w,*base,*wn;
   base=(complex_num*)malloc(sizeof(complex_num));
   if(lent==1)
   {
      base->real=coeff[0];
      base->next=NULL;
      return base;
   }
   int *coeff_o=new int[lent/2];
   int *coeff_e=new int[lent-lent/2];
   int e=0,o=0;
   w=(complex_num*)malloc(sizeof(complex_num));
   w->real=1;
   w->next=NULL;
   wn=(complex_num*)malloc(sizeof(complex_num));
   wn->real=cos(6.28/lent);
   wn->imag=sin(6.28/lent);
   wn->next=NULL;
   for(int i=0;i<lent;i++)
   {
      if(i%2==0)
      {
         coeff_e[e++]=coeff[i];
      }
      else
```

```
            {
               coeff_o[o++]=coeff[i];
            }
        }
        y1=fft(coeff_e,e);
        y2=fft(coeff_o,o);
        int k=lent/2;
        start1=(complex_num*)malloc(sizeof(complex_num));
        start1->next=NULL;
        start2=(complex_num*)malloc(sizeof(complex_num));
        start2->next=NULL;
        while(y1!=NULL)
        {
            if(y==NULL && mid==NULL)
            {
               y=start1;
               mid=start2;
            }
            else
            {
               complex_num *r=(complex_num*)malloc(sizeof(complex_num));
               y->next=r;
               y=y->next;
               y->next=NULL;
               mid->next=(complex_num*)malloc(sizeof(complex_num));
               mid=mid->next;
               mid->next=NULL;
            }
            complex_num* t=cal_mult(w,y2);
            y->real=y1->real+t->real;
            y->imag=y1->imag+t->imag;
            mid->real=(y1->real)-(t->real);
            mid->imag=(y1->imag)-(t->imag);
            w=cal_mult(w,wn);
            y1=y1->next;
            y2=y2->next;
        }
        y->next=start2;
        return start1;
}

int main()
{
        int *coeff;
        complex_num *res;
        int length,n;
        cout<<"\n Enter the value of N-point of Discrete-time (DT) signal x[n]\n ";
        cin>>length;
        n=length;
        while(n%2==0)
            n=n/2;
        if(n!=1)
        {
```

```
        cout<<"\nLength should be a power of 2\n";
        exit(0);
    }
    cout<<"\nEnter Values\n";
    coeff=new int[length];
    for(int i=0;i<length;i++)
    {
        cin>>coeff[i];
    }
    res=fft(coeff,length);
    cout<<"\n The N Complex discrete harmonics are\n";
    while(res!=NULL)
    {
        cout<<"\n"<<res->real<<" + "<<res->imag<<"i\n";
        res=res->next;
    }
    return 0;
}
```

Sample Output

```
prachi@prachi-Vostro-1014: ~
prachi@prachi-Vostro-1014:~$ g++ fft_copy.cpp
prachi@prachi-Vostro-1014:~$ ./a.out

 Enter the value of N-point of Discrete-time (DT) signal x[n]
 4

Enter Values
1
3
-1
-2

 The N Complex discrete harmonics are

1 + 0i

2.00398 + 5i

-1 + 0i

1.99602 + -5i
prachi@prachi-Vostro-1014:~$
```

Fig. 2 Output

```
prachi@prachi-Vostro-1014: ~
prachi@prachi-Vostro-1014:~$ g++ fft_copy.cpp
prachi@prachi-Vostro-1014:~$ ./a.out

 Enter the value of N-point of Discrete-time (DT) signal x[n]
 4

Enter Values
1
2
0
-1

 The N Complex discrete harmonics are

2 + 0i

1.00239 + 3i

0 + 0i

0.997611 + -3i
prachi@prachi-Vostro-1014:~$
```

Fig. 3 Output

```
prachi@prachi-Vostro-1014: ~
prachi@prachi-Vostro-1014:~$ g++ fft_copy.cpp
prachi@prachi-Vostro-1014:~$ ./a.out

 Enter the value of N-point of Discrete-time (DT) signal x[n]
 1

Enter Values
2

 The N Complex discrete harmonics are

2 + 0i
prachi@prachi-Vostro-1014:~$
```

Fig. 4 Output when the length of the input vector is 1

```
prachi@prachi-Vostro-1014: ~
prachi@prachi-Vostro-1014:~$ ./a.out

 Enter the value of N-point of Discrete-time (DT) signal x[n]
 23

Length should be a power of 2
prachi@prachi-Vostro-1014:~$
```

Fig. 5 Output when entered length of the input vector is not in power of 2

References

1. "Introduction to Algorithms", by Thomas H. Cormen
2. https://www.youtube.com/watch?v=EsJGuI7e_ZQ
3. http://www.csc.kth.se/utbildning/kth/kurser/DD2352/algokomp12/Forelasningar/F4.pdf
4. http://abut.sdsu.edu/TE302/Chap6.pdf