南開大學
Nankai University

南　　开　　大　　学

数 学 科 学 学 院

FFT 的程序实现与代码说明

李铭辉

年级：2020 级

专业：信息与计算科学

2023 年 5 月 28 日

摘要

关键字：FFT C/C++ 多线程 fftw3 库 混合基 基 2-DIT-FFT

目录

一、 背景

快速傅里叶算法是一种常用的计算傅里叶变换的算法。它通过利用傅里叶变换的对称性和周期性，将一个长度为 N 的离散序列的傅里叶变换分解成两个长度为 N/2 的序列的傅里叶变换，然后再继续对这两个长度为 N/2 的序列进行分解，最终获得长度为 1 的序列的傅里叶变换。这个过程可以用递归的方式实现。

快速傅里叶算法的时间复杂度为 $O(NlogN)$，比直接计算 $O(N^2)$ 更快。在数字信号处理、图像处理和音频处理等领域都有广泛应用。

为了实现高效的并行化 DIT-FFT，我将使用 pthread 技术，将计算任务分配给多个线程同时进行计算。由于 DIT-FFT 算法具有对称性和周期性，不同部分之间的计算可以并行进行，因此使用 pthread 技术可以大大提高计算效率。

为了验证我的算法的性能，我将与 fftw3 库进行性能比较。fftw3 是一个常用的高速傅里叶变换库，广泛应用于计算科学和工程中。通过与 fftw3 库进行比较，我可以评估我的算法在不同数据集和处理器实现中的性能表现，并找出性能瓶颈，进一步对算法进行改进。

为了进行性能比较，我将用随机生成的方法，生成不同长度（2 的幂次）的 double 数组构成数据集，对我的算法和 fftw3 进行计算时间的评测。

二、 fftw3 库

(一) 安装 fftw3 库

我使用的是 Ubuntu 系统，安装 fftw3 库只需在终端执行如下指令

```
sudo apt-get install libfftw3-dev
```

(二) fftw3 库的使用

使用 fftw3 库中的 FFT，需要：

1、使用 #include <fftw3.h> 引入库文件。

2、创建输入和输出数组来存储 FFT 的结果。输入数组存储需要计算 FFT 的数据，输出数组存储 FFT 的结果。数组的长度应该是 2 的幂次方。

3、创建 fftw_plan 对象并使用 fftw_plan_dft_1d 函数初始化该对象，该函数的参数为输入和输出数组以及 FFT 的大小。

4、调用 fftw_execute 函数执行 FFT 计算，并将结果存储在输出数组中。

5、最后，使用 fftw_destroy_plan 和 fftw_cleanup 函数释放对象和清空内存。

以下为代码示例

```
#include <fftw3.h>
#include <iostream>
using namespace std;
int main() {
    int N = 10;
    fftw_complex *in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*N);
    fftw_complex *out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*N);
    for (int i = 0; i < N; i++)
        cin >> in[i][0] >> in[i][1];
    fftw_plan p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
```

```
    fftw_execute(p);
    fftw_destroy_plan(p);
    fftw_cleanup();
    for (int i = 0; i < N; i++)
        cout << '(' << out[i][0] << ',' << out[i][1] << ')';
    fftw_free(in);
    fftw_free(out);
    return 0;
}
```

编译此代码，需要加上编译选项-lfftw3

三、 我的基 2-DIT-FFT

(一) 普通的基 2-DIT-FFT 算法

普通的基 2-DIT-FFT 算法的代码如下

```
int bitreverse(int x, int bits){
    int y = 0;
    for(int i=0; i<bits; i++){
        y = (y << 1) | (x & 1);
        x >>= 1;
    }
    return y;
}
complex<double>* base2_dit_fft(complex<double>* x, int N){
    int m = log2(N);
    complex<double>* x_1 = new complex<double>[N];
    for(int i=0; i<N; i++){
        int j = bitreverse(i,m);
        x_1[j] = x[i];
    }
    for(int i=1; i<=m; i++){
        int B = pow(2,i-1);
        for(int j=0; j<pow(2,m-i); j++){
            for(int k=0; k<pow(2,i-1); k++){
                double phi = -2*M_PI/pow(2,i)*k;
                complex<double> Wk = complex<double>(cos(phi),sin(phi));
                Wk *= x_1[k+j*int(pow(2,i))+B];
                x_1[k+j*int(pow(2,i))+B] = x_1[k+j*int(pow(2,i))] - Wk;
                x_1[k+j*int(pow(2,i))] = x_1[k+j*int(pow(2,i))] + Wk;
            }
        }
    }
    return x_1;
}
```

(二) 多线程设计

可以发现，在代码一中，有以下几处可以优化的内容：

1、第 12 行、第 18 行的 for 循环可以用多线程并行化；

2、第 25 行的复数 $cos(\phi) + sin(\phi)j$ 可以提前计算。

(三) 多线程 FFT 的关键代码

1. 多线程 FFT 的主要函数

```
#include <pthread.h>
int m; // 幂次数
int N; // N = 2^m 数组长度
double* x; // 输入数组
complex<double>* x_1; //输出数组
complex<double>* e; //存储e的幂次
const int THREAD_NUM = 2;// 线程数（至少为2）
typedef struct{ // 线程参数
    int r;
}threadParm_t;
pthread_barrier_t barrier;
void* fun_thread_1(void* parm);// 线程函数
void base2_dit_fft(){
//      线程初始化
    pthread_t thread[THREAD_NUM];
    threadParm_t threadParm[THREAD_NUM];
    pthread_barrier_init(&barrier,NULL,THREAD_NUM);
    for(int j=0;j<THREAD_NUM;j++){
        threadParm[j].r=j;
    }
//      启动线程
    for(int j=0;j<THREAD_NUM;j++){
        pthread_create(&thread[j],nullptr,fun_thread_1,(void*)&threadParm[j])
            ;
    }
//      等待线程
    for(int j=0;j<THREAD_NUM;j++){
        pthread_join(thread[j],nullptr);
    }
}
```

2. 线程函数的结构

```
void* fun_thread_1(void* parm){
    threadParm_t* p=(threadParm_t *) parm;
    int r=p->r;
    {
        ... // 第一步：给x换序
```

```
    }
    {
        ... // 第二步：计算e的幂次
    }
    pthread_barrier_wait(&barrier);
    {
        ... // 第三步：蝴蝶算法
    }
    pthread_exit(nullptr);
}
```

3. 换序算法的并行化

由于换序结果存放在另一个数组，所以换序过程互不冲突，可以直接按照输入数组 x 的长度均分任务，交给 THREAD_NUM 个线程完成换序。

```
{ // 给x换序
    // 分配任务
    int w = (N)/THREAD_NUM;
    int begin = r*w ;
    int last;
    if (r != THREAD_NUM-1)
        last = (r+1)*w ;
    else
        last = N;
    // 开始计算
    for(int i=begin; i<last; i++){
        int j = bitreverse(i,m);
        x_1[j] = x[i];
    }
}
```


4. e 幂次计算的并行化

注意到，在蝴蝶算法中，使用到的 e 的幂次有：

$$e^{-\pi\frac{0}{2^{1-1}}}, e^{-\pi\frac{0}{2^{2-1}}}, e^{-\pi\frac{1}{2^{2-1}}}, e^{-\pi\frac{0}{2^{3-1}}}, e^{-\pi\frac{1}{2^{3-1}}}, e^{-\pi\frac{2}{2^{3-1}}}, e^{-\pi\frac{3}{2^{3-1}}}, ..., e^{-\pi\frac{0}{2^{m-1}}}, e^{-\pi\frac{1}{2^{m-1}}}, ..., e^{-\pi\frac{2^{m-1}-1}{2^{m-1}}}, ...$$

2^m 长的数组的 FFT 只需要用到含 $e^{-\pi\frac{2^{m-1}-1}{2^{m-1}}}$ 及以前的 e 的幂次。共需要计算

$$2^{1-1} + 2^{2-1} + ... + 2^{m-1} = 2^m - 1$$

个 e 的幂次。将这些计算交给两个线程完成：

1、让 0 号线程计算

$$e^{-\pi\frac{0}{2^{1-1}}}, e^{-\pi\frac{0}{2^{2-1}}}, e^{-\pi\frac{1}{2^{2-1}}}, e^{-\pi\frac{0}{2^{3-1}}}, e^{-\pi\frac{1}{2^{3-1}}}, e^{-\pi\frac{2}{2^{3-1}}}, e^{-\pi\frac{3}{2^{3-1}}}, ..., e^{-\pi\frac{0}{2^{m-2}}}, e^{-\pi\frac{1}{2^{m-2}}}, ..., e^{-\pi\frac{2^{m-2}-1}{2^{m-2}}}$$

2、让 1 号线程计算

$$e^{-\pi\frac{0}{2^{m-1}}}, e^{-\pi\frac{1}{2^{m-1}}}, ..., e^{-\pi\frac{2^{m-1}-1}{2^{m-1}}}$$

如下是关键代码

```
{ // 计算e的幂次
    if (r == 0){ // 0号线程的计算
        int B=1;
        for (int i = 1; i <= m-1; i++) {
            for(int k= 0; k<B ; k++)
                e[B+k-1] = complex<double>( cos(-(M_PI/B*k)),sin(-(M_PI/B*k))
                    ) ;
            B = B << 1;
        }
    }
    if (r == 1){ // 1号线程的计算
        int B = pow(2,m-1);
        for(int k= 0; k<B ; k++)
            e[B+k-1] = complex<double>( cos(-(M_PI/B*k)),sin(-(M_PI/B*k)) ) ;
    }
}
```

5. 蝴蝶算法的并行化

1、仔细分析普通的 fft 代码一，发现第 16 行的 for 循环中的计算内容与 i 有关，也就是说下一轮计算要用到上一轮的计算结果，即存在数据依赖。因此并行化实现时，需要在此循环中同步线程。

2、在循环中，发现有多次 2 的幂次计算（例如：$pow(2, i-1)$、$pow(2, m-i)$）是不必要的，可以用循环中的移位运算替代。

如下是关键代码

```
{ // 蝴蝶算法
    complex<double> Wk ;
    int B = 1;
    int tempINT = pow(2,m-1);
    for(int i=1; i<=m; i++){
        // 分配任务
        int w = (tempINT)/THREAD_NUM;
        int begin = r * w ;
        int last;
        if (r != THREAD_NUM-1)
            last = begin + w ;
        else
            last = tempINT;
        // 开始计算
        for(int j=begin; j<last; j++){
            int tempINT2 = j * ( B<<1 );
            for(int k=0; k<B; k++){
                Wk = e[B+k-1] * x_1[tempINT2+B];
                x_1[tempINT2+B] = x_1[tempINT2] - Wk;
                x_1[tempINT2] += Wk;
                tempINT2++;
```

```
                }
            }
            // 通过移位来计算2的幂次
            B = B << 1;
            tempINT = tempINT >> 1;
            pthread_barrier_wait(&barrier);
        }
    }
```

6. 编译选项

编译选项包含：-O3、-lpthread

(四) 多线程 FFT 程序

1. C++ 代码

我的多线程 FFT 程序的 C++ 代码请看github。

2. 使用方法

第一步、编译 C++ 代码

```
g++ ./myfft.cpp -O3 -o myfft -lpthread
```

第二步、运行 fft 程序

```
./myfft 8 in.txt out.txt
```

运行 fft 程序共需要 3 个参数:
第一个是正整数 m，表示数组长度为 2^m。
第二个是输入数组所在文件的名称 in.txt。2^m 个 double 型数据按照一行存储其中，间隔符为空格。
第三个是输出数组所在文件的名称 out.txt。2^{m+1} 个 double 型数据按照一行存储其中，间隔符为空格。傅里叶变换结果中第 i 个复数的实部是 out.txt 中的第 2i 个 double 型数据，虚部是 out.txt 中第 2i+1 个 double 型数据。

四、 我的混合基 FFT

(一) 混合基 FFT 的原理

原理参见 [博客](https://zjuturtle.com/2017/12/26/fft/)。但是由于我还没有想明白如何把这里头的递归展开成普通循环，也暂时想不到行之有效的并行化加速方法。所以只是用 C++ 代码原原本本地实现了混合基 FFT 算法。速度上，比离散傅里叶变换快许多，但是和我上一步实现的基 2-DIT-FFT 算法以及 fftw3 库中的 FFT 算法相比，落后很多。因此就不浪费时间计时了。

名称	Intel(R) Celeron(R) CPU J1800
基准频率	2.41 GHz
内核数	2
逻辑处理器	4
L1d 缓存	48KB
L1i 缓存	64KB
L2 缓存	1MB

表 1: CPU 信息

(二) 混合基 FFT 程序

1. C++ 代码

我的混合基 FFT 程序的 C++ 代码请看github。

2. 使用方法

第一步、编译 C++ 代码

```
g++ mixed_radix_FFT.cpp -O3 -o mixed_radix_FFT
```

第二步、运行混合基 FFT 程序

```
./mixed_radix_FFT 12 in.txt out.txt
```

运行混合基 FFT 程序共需要 3 个参数:
第一个是正整数 N，表示数组长度为 12。
第二个是输入数组所在文件的名称 in.txt。2^m 个 double 型数据按照一行存储其中，间隔符为空格。
第三个是输出数组所在文件的名称 out.txt。2^{m+1} 个 double 型数据按照一行存储其中，间隔符为空格。傅里叶变换结果中第 i 个复数的实部是 out.txt 中的第 2i 个 double 型数据，虚部是 out.txt 中第 2i+1 个 double 型数据。

五、 性能比较结果

(一) 实验设备

操作系统：Ubuntu 20.04.6 LTS (GNULinux 5.15.0-71-generic x86_64)
内存容量：大小 8GB，速度 1333MHz
编译器版本：g++ (Ubuntu 9.4.0-1ubuntu1 20.04.1) 9.4.0
编译器设置：-lpthread、-lfftw3、-O3
CPU 信息如表1所示。

(二) 测试方法以及实验结果

1. 测试数据

测试规模有：2^2,2^3,2^4,...,2^{20}。每种规模的矩阵生成 10 份。使用 C++ 常见的 rand() 来生成。

```
time_t t;
srand((unsigned)time(&t));
for (int i = 0; i < N; i++) {
    x[i] = (double)rand() / (double)RAND_MAX; // 生成0到1之间的随机double数
}
```

2. 计时程序

使用 C++ 的 time 库来计时，计时程序的关键代码如下

```
double time = 0;
for(int i=0;i<10;i++)
{
    ...// 读取该规模下的不同数组，作为输入
    struct timespec sts,ets;
    timespec_get(&sts, TIME_UTC);

    ...// 计时内容

    timespec_get(&ets, TIME_UTC);
    time_t dsec=ets.tv_sec-sts.tv_sec;
    long dnsec=ets.tv_nsec-sts.tv_nsec;
    if (dnsec<0){
        dsec--;
        dnsec+=1000000000ll;
    }
    time += (dsec + dnsec * 1e-9);
}
time /=10;
printf ("%lf\n",time); // 输出计时结果
```

3. 计时内容

myfft 的计时内容为

```
base2_dit_fft();
```

fftw3 的计时内容为

```
fftw_plan p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD,FFTW_ESTIMATE);
fftw_execute(p);
```

4. 计时方法

每运行一次计时程序，就可以得到 fft 算法在指定规模下，对于 10 个数组的平均运行时间。为了消除环境因素的影响，针对每种规模，计时程序会运行 10 次，并取平均值作为结果。

5. 计时结果

如表格2所示。

2 的幂次	fftw3 用时（单位:s）	myfft 用时（单位:s）	myfft 用时/fftw3 用时
2	0.0004841	0.0002121	0.438132617
3	0.0004648	0.0001999	0.430077453
4	0.0004792	0.0002194	0.457846411
5	0.0005721	0.0002319	0.405348715
6	0.0005165	0.0003365	0.651500484
7	0.0005583	0.0002762	0.494716102
8	0.0005877	0.0004319	0.734898758
9	0.0006166	0.0003575	0.57979241
10	0.0006413	0.0004628	0.72165913
11	0.0007778	0.0013335	1.714451016
12	0.0009411	0.0010498	1.115503135
13	0.0014038	0.0016672	1.187633566
14	0.0020656	0.0036552	1.769558482
15	0.0036779	0.0066837	1.817259849
16	0.0085697	0.0153729	1.793866763
17	0.0174431	0.0318146	1.823907448
18	0.036068	0.065079	1.804341799
19	0.089062	0.128842	1.446655139
20	0.18218	0.271394	1.489702492
21	0.375512	0.564445	1.503134387
22	0.78297	1.184179	1.512419377
23	1.678055	2.347761	1.399096573

表 2: 时间对比

(三) 结果分析

1、受限于机能，强大的 fftw3 库似乎也无能为力。以 2^{20} 长数组为例，在数院的服务器的平均用时在 20ms 以内（来自于同学的测试），而我这个机器却在 182ms 左右，差距甚大。

2、在数组长度不大时，我编写的 fft 用时比 fftw3 库的 fft 短，这是因为它有额外的一步——制定 plan。在数组长度较大时，我编写的 fft 用时与 fftw3 库的 fft 差距不大，这是因为机器性能太差。

3、没有进行更大规模数组计时的原因：存取数组的耗时太长，占用空间太大。即便不采取读取的办法，在计时前生成数组，也会因为数组过长而内存不够。这方面我懂得太少，所以就没有深入优化了。