

实验报告

实验名称（测量 FFT 程序执行时间）

智能 1602 201608010609 李鹏飞

实验目标

测量 FFT 程序运行时间，确定其时间复杂度。

实验要求

- * 采用 C/C++编写程序
- * 根据自己的机器配置选择合适的输入数据大小 n ，至少要测试多个不同的 n (参见思考题)
- * 对于相同的 n ，建议重复测量 30 次取平均值作为测量结果 (参见思考题)
- * 对测量结果进行分析，确定 FFT 程序的时间复杂度
- * 回答思考题，答案加入到实验报告叙述中合适位置

思考题

1. 分析 FFT 程序的时间复杂度，得到执行时间相对于数据规模 n 的具体公式
2. 根据上一点中的分析，至少要测试多少不同的 n 来确定执行时间公式中的未知数？
3. 重复 30 次测量然后取平均有什么统计学的依据？

实验内容

FFT 算法代码（数据规模 n 为 64，循环 30 次）

FFT 的算法如下，在这里使用了递归的方法：

```
/* fft.cpp
*
* This is a KISS implementation of
* the Cooley-Tukey recursive FFT algorithm.
* This works, and is visibly clear about what is happening where.
*
```

```

* To compile this with the GNU/GCC compiler:
* g++ -o fft fft.cpp -lm
*
* To run the compiled version from a *nix command line:
* ./fft
*
*/
#include <complex>
#include <cstdio>
#include <iostream>
#include <ctime>

#define M_PI 3.14159265358979323846 // Pi constant with double precision

using namespace std;

// separate even/odd elements to lower/upper halves of array respectively.
// Due to Butterfly combinations, this turns out to be the simplest way
// to get the job done without clobbering the wrong elements.
void separate (complex<double>* a, int n) {
    complex<double>* b = new complex<double>[n/2]; // get temp heap storage
    for(int i=0; i<n/2; i++) // copy all odd elements to heap storage
        b[i] = a[i*2+1];
    for(int i=0; i<n/2; i++) // copy all even elements to lower-half of a[]
        a[i] = a[i*2];
    for(int i=0; i<n/2; i++) // copy all odd (from heap) to upper-half of a[]
        a[i+n/2] = b[i];
    delete[] b; // delete heap storage
}

// N must be a power-of-2, or bad things will happen.
// Currently no check for this condition.
//
// N input samples in X[] are FFT'd and results left in X[].
// Because of Nyquist theorem, N samples means
// only first N/2 FFT results in X[] are the answer.
// (upper half of X[] is a reflection with no new information).
void fft2 (complex<double>* X, int N) {
    if(N < 2) {
        // bottom of recursion.
        // Do nothing here, because already X[0] = x[0]
    } else {

```

```

separate(X,N); // all evens to lower half, all odds to upper half
fft2(X, N/2); // recurse even items
fft2(X+N/2, N/2); // recurse odd items
// combine results of two half recursions
for(int k=0; k<N/2; k++) {
    complex<double> e = X[k ]; // even
    complex<double> o = X[k+N/2]; // odd
    // w is the "twiddle-factor"
    complex<double> w = exp( complex<double>(0,-2.*M_PI*k/N) );
    X[k ] = e + w * o;
    X[k+N/2] = e - w * o;
}
}
}

// simple test program
int main () {
    clock_t start,finish;
    start = clock();
    for(int z=0;z<30;z++){
        const int nSamples = 64;
        double nSeconds = 1.0; // total time for sampling
        double sampleRate = nSamples / nSeconds; // n Hz = n / second
        double freqResolution = sampleRate / nSamples; // freq step in FFT result
        complex<double> x[nSamples]; // storage for sample data
        complex<double> X[nSamples]; // storage for FFT answer
        const int nFreqs = 5;
        double freq[nFreqs] = { 2, 5, 11, 17, 29 }; // known freqs for testing
        // generate samples for testing
        for(int i=0; i<nSamples; i++) {
            x[i] = complex<double>(0.,0.);
            // sum several known sinusoids into x[]
            for(int j=0; j<nFreqs; j++)
                x[i] += sin( 2*M_PI*freq[j]*i/nSamples );
            X[i] = x[i]; // copy into X[] for FFT work & result
        }
        // compute fft for this data
        fft2(X,nSamples);
        printf(" n\tx[]\tX[]\tf\n"); // header line
        // loop to print values
        for(int i=0; i<nSamples; i++) {
            printf("% 3d\t%+.3f\t%+.3f\t%g\n",
                i, x[i].real(), abs(X[i]), i*freqResolution );
        }
        finish = clock();
    }
}

```

```
cout<<(finish - start) <<"/"<<CLOCKS_PER_SEC*30 << "(s)"<<endl;
return 0;
}

// eof
```

FFT 程序时间复杂度分析

通过分析 FFT 算法代码，可以得到该 FFT 算法的时间复杂度具体公式为：

$$a*n+3/2*b*n*\log n+c$$

其中*n*为数据大小，未知数有：

1. *a*
2. *b*
3. *c*

通过化简我们可以得到 FFT 算法时间复杂度为 $O(n\log n)$

测试

测试平台

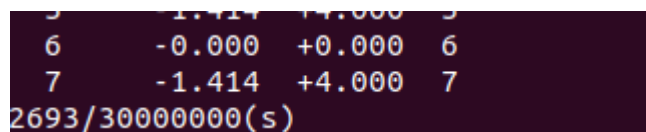
在如下机器上进行了测试：

| 部件 | 配置 | 备注 |
|------|------------------|-----|
| CPU | core i5-6300HQ | |
| 内存 | DDR4 12GB | |
| 操作系统 | Ubuntu 18.04 LTS | 中文版 |

测试记录

测试不同规模的 FFT 运行时间(循环 30 次)

* n=8



```

5      1.414  +4.000  5
6      -0.000  +0.000  6
7      -1.414  +4.000  7
2693/300000000(s)
```

* n=16

```
13      -0.955  +8.000  13
14      -1.414  +8.000  14
15      -3.721  +8.000  15
3642/30000000(s)
```

* n=32

```
26      +1.873  +0.000  26
27      -0.861  +0.000  27
28      -1.414  +16.000 28
29      -2.785  +16.000 29
30      -0.892  +0.000  30
31      -2.764  +16.000 31
5778/30000000(s)
```

* n=64

```
58      +0.222  +0.000  58
59      -2.570  +32.000  59
60      -0.166  +0.000  60
61      -1.269  +0.000  61
62      -1.295  +32.000  62
63      -2.834  +0.000  63
10142/30000000(s)
```

利用 perf 工具追踪程序运行情况

```
Performance counter stats for './FFT':

      8.952648      task-clock (msec)      #    0.570 CPUs utilized
           1      context-switches        #    0.112 K/sec
           0      cpu-migrations          #    0.000 K/sec
        125      page-faults              #    0.014 M/sec
  24,103,987      cycles                   #    2.692 GHz
  33,588,111      instructions             #    1.39  insn per cycle
   6,168,491      branches                 #   689.013 M/sec
    87,846      branch-misses              #    1.42% of all branches

0.015720130 seconds time elapsed
```

思考题解答

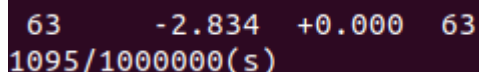
1.测试次数

在这里我们的复杂度具体公式设立了三个未知数，为了求解三个未知数的值，我们在这里需要至少 3 个 n 才能求解

2.统计学依据

在这里我们用到了累积法测量的思想：

因为利用 FFT 提高代码效率，这就使得单次运行时间很短，例如下图：



```
63      -2.834   +0.000   63
1095/1000000(s)
```

我们看到单次时间很短，容易出现误差，所以我们做 30 次取平均减小误差。

分析和结论

从测试记录来看，FFT 程序的执行时间随数据规模增大而增大，其时间复杂度为 $O(n \log n)$ 。在一开始单一运行的时候计算得时间与计算 30 次之后取平均的值有较大差异，这充分说明了我们在进行研究的时候要注意多次取平均以减小误差。