# 实验报告

## 实验名称（多线程 FFT 程序性能分析和测试）

智能 1501 201508010528 耿昊

## 实验目标

测量多线程 FFT 程序运行时间，考察线程数目增加时运行时间的变化。

## 实验要求

- 采用 C/C++编写程序，选择合适的运行时间测量方法
- 根据自己的机器配置选择合适的输入数据大小 n，保证足够长度的运行时间
- 对于不同的线程数目，建议至少选择 1 个，2 个，4 个，8 个，16 个线程进行测试
- 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.pthread 是什么？怎么使用？

2.多线程相对于单线程理论上能提升多少性能？多线程的开销有哪些？

3.实际运行中多线程相对于单线程是否提升了性能？与理论预测相差多少？可能的原因是什么？

## 实验内容

### 多线程 FFT 代码

```
/ Threaded two-dimensional Discrete FFT transform
// Guru Das Srinagesh
// ECE6122 Project 2


#include <iostream>
#include <string>
#include <math.h>

#include "Complex.h"
#include "InputImage.h"

#include <stdio.h>
#include <pthread.h>

// You will likely need global variables indicating how
// many threads there are, and a Complex* that points to the
// 2d image being transformed.

Complex* ImageData;
int ImageWidth;
int ImageHeight;

#define N_THREADS 16

#define FORWARD   1
#define INVERSE  -1

int inverse = FORWARD;
```

```
int N = 1024;                 // Number of points in the 1-D transform

/* pThreads variables */
pthread_mutex_t exitMutex;    // For exitcond
pthread_mutex_t printfMutex;  // Not sure if mutex is reqd for printf
pthread_cond_t  exitCond;     // Project req demands its existence

Complex* W;                   // Twiddle factors

/* Variables for MyBarrier */
int             count;        // Number of threads presently in the barrier
pthread_mutex_t countMutex;
bool*           localSense;   // We will create an array of bools, one per thread
bool            globalSense;  // Global sense

using namespace std;

// Function to reverse bits in an unsigned integer
// This assumes there is a global variable N that is the
// number of points in the 1D transform.
unsigned ReverseBits(unsigned v)
{ //  Provided to students
  unsigned n = N; // Size of array (which is even 2 power k value)
  unsigned r = 0; // Return value

  for (--n; n > 0; n >>= 1)
    {
      r <<= 1;        // Shift return value
      r |= (v & 0x1); // Merge in next bit
      v >>= 1;        // Shift reversal value
    }
  return r;
}

// GRAD Students implement the following 2 functions.
// Call MyBarrier_Init once in main
void MyBarrier_Init()// you will likely need some parameters)
{
  count = N_THREADS + 1;

  /* Initialize the mutex used for MyBarrier() */
  pthread_mutex_init(&countMutex, 0);

  /* Create and initialize the localSense array, 1 entry per thread */
  localSense = new bool[N_THREADS + 1];
  for (int i = 0; i < (N_THREADS + 1); ++i) localSense[i] = true;

  /* Initialize global sense */
  globalSense = true;
}

int FetchAndDecrementCount()
{
  /* We don’t have an atomic FetchAndDecrement, but we can get the */
  /* same behavior by using a mutex */

  pthread_mutex_lock(&countMutex);
  int myCount = count;
  count--;
  pthread_mutex_unlock(&countMutex);
  return myCount;
}

// Each thread calls MyBarrier after completing the row-wise DFT
void MyBarrier(unsigned threadId)
{
  localSense[threadId] = !localSense[threadId]; // Toggle private sense variable
  if (FetchAndDecrementCount() == 1)
  { // All threads here, reset count and toggle global sense
```

```
    count = N_THREADS+1;
    globalSense = localSense[threadId];
  }
  else
  {
    while (globalSense != localSense[threadId]) { } // Spin
  }
}

void precomputeW(int inverse)
{
  W = new Complex[ImageWidth];

  /* Compute W only for first half */
  for(int n=0; n<(ImageWidth/2); n++){
    W[n].real = cos(2*M_PI*n/ImageWidth);
    W[n].imag = -inverse*sin(2*M_PI*n/ImageWidth);
  }
}

void Transform1D(Complex* h, int N)
{
  // Implement the efficient Danielson-Lanczos DFT here.
  // "h" is an input/output parameter
  // "N" is the size of the array (assume even power of 2)

  /* Reorder array based on bit reversing */
  for(int i=0; i<N; i++){
    int rev_i = ReverseBits(i);
    if(rev_i < i){
      Complex temp = h[i];
      h[i] = h[rev_i];
      h[rev_i] = temp;
    }
  }

  /* Danielson-Lanczos Algorithm */
  for(int pt=2; pt <= N; pt*=2)
    for(int j=0; j < (N); j+=pt)
      for(int k=0; k < (pt/2); k++){
        int offset = pt/2;
        Complex oldfirst = h[j+k];
        Complex oldsecond = h[j+k+offset];
        h[j+k] = oldfirst + W[k*N/pt]*oldsecond;
        h[j+k+offset] = oldfirst - W[k*N/pt]*oldsecond;
      }

  if(inverse == INVERSE){
    for(int i=0; i<N; i++){
      // If inverse, then divide by N
      h[i] = Complex(1/(float)(N))*h[i];
    }
  }
}

void* Transform2DTHread(void* v)
{ // This is the thread starting point.  "v" is the thread number
  // Calculate 1d DFT for assigned rows
  // wait for all to complete
  // Calculate 1d DFT for assigned columns
  // Decrement active count and signal main if all complete

  /* Determine thread ID */
  unsigned long thread_id = (unsigned long)v;

  /* Determine starting row and number of rows per thread */
  int rowsPerThread = ImageHeight / N_THREADS;
  int startingRow = thread_id * rowsPerThread;

  for(int row=startingRow; row < (startingRow + rowsPerThread); row++){
```

```
    Transform1D(&ImageData[row * ImageWidth], N);
  }

  pthread_mutex_lock(&printfMutex);
  printf("  Thread %2ld: My part is done! \n", thread_id);
  pthread_mutex_unlock(&printfMutex);

  /* Call barrier */
  MyBarrier(thread_id);

  /* Trigger cond_wait */
  if(thread_id == 5){
    pthread_mutex_lock(&exitMutex);
    pthread_cond_signal(&exitCond);
    pthread_mutex_unlock(&exitMutex);
  }

  return 0;
}

void Transform2D(const char* inputFN)
{
  /* Do the 2D transform here. */

  InputImage image(inputFN);        // Read in the image
  ImageWidth = image.GetWidth();
  ImageHeight = image.GetHeight();

  // All mutex and condition variables must be initialized
  pthread_mutex_init(&exitMutex,0);
  pthread_mutex_init(&printfMutex,0);
  pthread_cond_init(&exitCond, 0);

  // Create the global pointer to the image array data
  ImageData = image.GetImageData();

  // Precompute W values
  precomputeW(FORWARD);

  // Hold the exit mutex until waiting for exitCond condition
  pthread_mutex_lock(&exitMutex);

  /* Init the Barrier stuff */
  MyBarrier_Init();

  /* Declare the threads */
  pthread_t threads[N_THREADS];

  int i = 0;  // The humble omnipresent loop variable

  // Create 16 threads
  for(i=0; i < N_THREADS; ++i){
    pthread_create(&threads[i], 0, Transform2DTHread, (void *)i);
  }

  // Write the transformed data
  image.SaveImageData("MyAfter1d.txt", ImageData, ImageWidth, ImageHeight);
  cout<<"\n1-D transform of Tower.txt done"<<endl;
  MyBarrier(N_THREADS);

  /* Transpose the 1-D transformed image */
  for(int row=0; row<N; row++)
    for(int column=0; column<N; column++){
      if(column < row){
        Complex temp; temp = ImageData[row*N + column];
        ImageData[row*N + column] = ImageData[column*N + row];
        ImageData[column*N + row] = temp;
      }
    }
  cout<<"Transpose done"<<endl<<endl;
```

```
// /* -------- */  startCount = N_THREADS;
/* Do 1-D transform again */
// Create 16 threads
for(i=0; i < N_THREADS; ++i){
  pthread_create(&threads[i], 0, Transform2DTHread, (void *)i);
}

// Wait for all threads complete
MyBarrier(N_THREADS);
pthread_cond_wait(&exitCond, &exitMutex);

/* Transpose the 1-D transformed image */
for(int row=0; row<N; row++)
  for(int column=0; column<N; column++){
    if(column < row){
      Complex temp; temp = ImageData[row*N + column];
      ImageData[row*N + column] = ImageData[column*N + row];
      ImageData[column*N + row] = temp;
    }
  }
cout<<"\nTranspose done"<<endl;

// Write the transformed data
image.SaveImageData("Tower-DFT2D.txt", ImageData, ImageWidth, ImageHeight);
cout<<"2-D transform of Tower.txt done"<<endl<<endl;

//------------------------------------------------------------------------
//------------------------------------------------------------------------

/* Calculate Inverse */

// Precompute W values
precomputeW(INVERSE);
inverse = INVERSE;
// /* -------- */  startCount = N_THREADS;
/* Do 1-D transform again */
// Create 16 threads
for(i=0; i < N_THREADS; ++i){
  pthread_create(&threads[i], 0, Transform2DTHread, (void *)i);
}

// Wait for all threads complete
MyBarrier(N_THREADS);
pthread_cond_wait(&exitCond, &exitMutex);

/* Transpose the 1-D transformed image */
for(int row=0; row<N; row++)
  for(int column=0; column<N; column++){
    if(column < row){
      Complex temp; temp = ImageData[row*N + column];
      ImageData[row*N + column] = ImageData[column*N + row];
      ImageData[column*N + row] = temp;
    }
  }
cout<<"\nTranspose done\n"<<endl;

// /* -------- */  startCount = N_THREADS;
/* Do 1-D transform again */
// Create 16 threads
for(i=0; i < N_THREADS; ++i){
  pthread_create(&threads[i], 0, Transform2DTHread, (void *)i);
}

// Wait for all threads complete
MyBarrier(N_THREADS);
pthread_cond_wait(&exitCond, &exitMutex);

/* Transpose the 1-D transformed image */
for(int row=0; row<N; row++)
```

```
    for(int column=0; column<N; column++){
      if(column < row){
        Complex temp; temp = ImageData[row*N + column];
        ImageData[row*N + column] = ImageData[column*N + row];
        ImageData[column*N + row] = temp;
      }
    }
  cout<<"\nTranspose done"<<endl;

  // Write the transformed data
  image.SaveImageData("MyAfterInverse.txt", ImageData, ImageWidth, ImageHeight);
  cout<<"2-D inverse of Tower.txt done\n"<<endl;
}

int main(int argc, char** argv)
{
  string fn("Tower.txt");            // default file name

  if (argc > 1) fn = string(argv[1]);   // if name specified on cmd line

  Transform2D(fn.c_str());            // Perform the transform.
}
```

该代码采用了 pthread 库来实现多线程，

**POSIX 线程**（POSIX threads），简称 Pthreads，是线程的 **POSIX 标准**。该标准定义了创建和操纵线程的一整套 API。在**类 Unix 操作系统**（Unix、Linux、Mac OS X 等）中，都使用 Pthreads 作为操作系统的线程。

# 数据类型

pthread_t：线程 ID

pthread_attr_t：线程属性

# 操纵函数

pthread_create()：创建一个线程

pthread_exit()：终止当前线程

pthread_cancel()：中断另外一个线程的运行

pthread_join()：阻塞当前的线程，直到另外一个线程运行结束

pthread_attr_init()：初始化线程的属性

pthread_attr_setdetachstate()：设置脱离状态的属性（决定这个线程在终止时是否可以被结合）

pthread_attr_getdetachstate()：获取脱离状态的属性

pthread_attr_destroy()：删除线程的属性

pthread_kill()：向线程发送一个信号

# 同步函数

用于 mutex 和条件变量

pthread_mutex_init() 初始化互斥锁

pthread_mutex_destroy() 删除互斥锁

pthread_mutex_lock()：占有互斥锁（阻塞操作）
pthread_mutex_trylock()：试图占有互斥锁（不阻塞操作）。即，当互斥锁空闲时，将占有该锁；否则，立即返回。
pthread_mutex_unlock(): 释放互斥锁
pthread_cond_init()：初始化条件变量
pthread_cond_destroy()：销毁条件变量
pthread_cond_signal(): 唤醒第一个调用 pthread_cond_wait()而进入睡眠的线程
pthread_cond_wait(): 等待条件变量的特殊条件发生
Thread-local storage（或者以 Pthreads 术语，称作*线程特有数据*）：
pthread_key_create(): 分配用于标识进程中线程特定数据的键
pthread_setspecific(): 为指定线程特定数据键设置线程特定绑定
pthread_getspecific(): 获取调用线程的键绑定，并将该绑定存储在 value 指向的位置中
pthread_key_delete(): 销毁现有线程特定数据键
pthread_attr_getschedparam();获取线程优先级
pthread_attr_setschedparam();设置线程优先级

# 工具函数

pthread_equal(): 对两个线程的线程标识号进行比较
pthread_detach(): 分离线程
pthread_self(): 查询线程自身线程标识号

其中，本次我们主要用到了互斥量。

互斥量：Mutex
a.　用于互斥访问
b.　类型：pthread_mutex_t，必须被初始化为 PTHREAD_MUTEX_INITIALIZER（用于静态分配的 mutex，等价于 pthread_mutex_init(…, NULL)）或者调用 pthread_mutex_init。Mutex 也应该用 pthread_mutex_destroy 来销毁。这两个函数原型如下：（attr 的具体含义下一章讨论）

```
＃i nclude <pthread.h>

int pthread_mutex_init(
      pthread_mutex_t *restrict mutex,
      const pthread_mutexattr_t *restrict attr)

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

c.　pthread_mutex_lock 用于 Lock Mutex，如果 Mutex 已经被 Lock，该函数调用会 Block 直到 Mutex 被 Unlock，然后该函数会 Lock Mutex 并返回。

pthread_mutex_trylock 类似，只是当 Mutex 被 Lock 的时候不会 Block，而是返回一个错误值 EBUSY。 pthread_mutex_unlock 则是 unlock 一个 mutex。这三个函数原型如下：

```
＃i nclude <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

### 多线程 FFT 程序性能分析

通过对多线程 FFT 程序代码的分析,我们可以看到,该程序的多个线程需要向同一个文件写入输出数据,各个线程彼此的影响可以基本忽略,因为各个线程基本不可能同时完成计算并需要写数据,所以不存在同时需要执行写操作而造成的空闲等待时间误差。所以可以认为各个线程是原任务的完全相等的子问题。在进行 n 线程并发的情况下,理论运算时间应该是单线程的 ,也就是说理论加速比=线程并发数 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 | Intel® Core™ i5-8300H CPU @ 2.30GHz | 虚拟机 |
| 内存 | DDR4 6GB | |
| 操作系统 | Ubuntu 18.10LTS | 中文版 |

### 测试记录

多线程 FFT 程序的测试参数如下：

| 参数 | 取值 | 备注 |
|---|---|---|
| 数据规模 | 1024 或其它 | |
| 线程数目 | 1,2,4,8,16,32 | |

多线程 FFT 程序运行过程的截图如下：

```
 Thread  9: My part is done!
 Thread  5: My part is done!
 Thread  4: My part is done!
 Thread  0: My part is done!
 Thread  6: My part is done!
 Thread 14: My part is done!
 Thread 10: My part is done!
 Thread  8: My part is done!
 Thread 15: My part is done!
 Thread  7: My part is done!
 Thread  3: My part is done!
 Thread 11: My part is done!
 Thread  2: My part is done!
 Thread 13: My part is done!
 Thread  1: My part is done!

ranspose done
-D inverse of Tower.txt done


eal     0m11.474s
ser     0m10.747s
ys      0m0.067s
```

16:

```
Transpose done

  Thread  3: My part is done!
  Thread  6: My part is done!
  Thread  2: My part is done!
  Thread  7: My part is done!
  Thread  4: My part is done!
  Thread  0: My part is done!
  Thread  5: My part is done!
  Thread  1: My part is done!

Transpose done
2-D inverse of Tower.txt done


real    0m6.873s
user    0m6.583s
sys     0m0.047s
```

8:

```
Transpose done

  Thread  3: My part is done!
  Thread  2: My part is done!
  Thread  0: My part is done!
  Thread  1: My part is done!

Transpose done
2-D inverse of Tower.txt done


real    0m5.051s
user    0m4.684s
sys     0m0.092s
```

4:

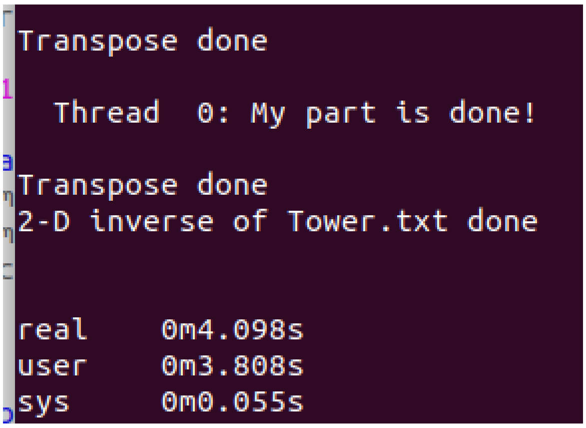
```
Transpose done

  Thread  0: My part is done!
  Thread  1: My part is done!

Transpose done
2-D inverse of Tower.txt done


real    0m4.170s
user    0m3.905s
sys     0m0.050s
```

2:

1:



FFT 程序的输出

根据计算 1、2、4、8、16 线程下 fft 程序运行时间，可以算出每种情况下单个线程需要的时间：

| 线程数 | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 总时间 | 4.1 | 4.2 | 5.1 | 6.9 | 11.4 |
| 单个线程时间 | 4.1 | 2.1 | 1.3 | 0.86 | 0.71 |

# 分析和结论

从测试记录来看，FFT 程序的执行时间随线程数目增大而增大，其相对于单线程情况的加速比分别为

| 线程数 | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 对单线程加速比 | 1 | 2 | 3.2 | 4.8 | 5.7 |

实际来看，多线程确实比单线程提高了性能，但是和理论相比，多线程程序的加速比不如预期。

比如 4 线程就有 20%误差，8 线程有 40%误差

原因在于

多线程的开销主要有:
1. 线程的创建以及撤销的时间开销
2. 每个线程独立的寄存器,栈,程序计数器,内容等空间开销
3. 线程间进行上下文切换需要额外的时间
4. 线程发生阻塞的时间