实验报告

实验名称(多线程 FFT 程序性能分析和测试)

智能 1501 201508010513 王鑫淼

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

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

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

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

```
#include <time.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.
}
```

本函数中使用了 pthread，pthread 是 POSIX thread 的简称，该标准定义内部 API 创建和操纵线程，Pthreads 定义了一套 C 程序语言类型、函数与常量，它以 pthread.h 头文件和一个线程库实现。

这个头文件中定义了很多使用 pthread 的方便函数，在本次实验中使用的有：

pthread_mutex_t：互斥锁类型

pthread_cond_t：条件变量类型

pthread_create()：创建一个线程

pthread_mutex_init() 初始化互斥锁

pthread_mutex_lock()：占有互斥锁（阻塞操作）

pthread_mutex_unlock(): 释放互斥锁

pthread_cond_init()：初始化条件变量

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

互斥锁：在编程中，引入了对象互斥锁的概念，来保证共享数据操作的完整性。每个对象都对应于一个可称为"互斥锁" 的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。有静态方式和动态方式两种方法创建互斥锁。互斥锁的属性在创建锁的时候指定，在 LinuxThreads 实现中仅有一个锁类型属性，不同的锁类型在试图对一个已经被锁定的互斥锁加锁时表现不同。

条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起。

多线程 FFT 程序性能分析

根据以上多线程 FFT 程序代码，可以看到多线程是并发进行的，各个线程同时运行，忽略其各个线程之间的切换间隔，可以知道多个线程中每个线程的运行时间基本上是相同的，那么在有 N_thread 个线程的情况下，加速比（多线程：单线程）= N_thread。

根据代码，可以看到在非线程运行部分，有多个对矩阵进行转置或其他操作，其中矩阵的规模为 n（1024），这个部分的运行时间占据了 FFT 程序的运行时间的大部分，其时间复杂度为 $O(n^2)$，而线程部分由于所有线程的运行时间与单个线程运行时间基本相同，与线程数量无关，其时间复杂度为 $O(n^2+nlog^2n)$即 $O(n^2)$；最后将数据保存到文件的操作器时间复杂度同样为 $O(n^2)$；所以 FFT 程序的时间复杂度为 $O(a*n^2+b*nlog^2n+c*n+d)$即 $O(n^2)$。

测试

测试平台

在如下机器上进行了测试：

部件	配置
cpu	核心 I5-5200 U
内存	DDR 3 1GB
操作系统	Ubuntu 17.04 LTS

测试记录

多线程 FFT 程序的测试参数如下：

参数	取值
数据规模	1024
线程数目	1,2,4,8,16

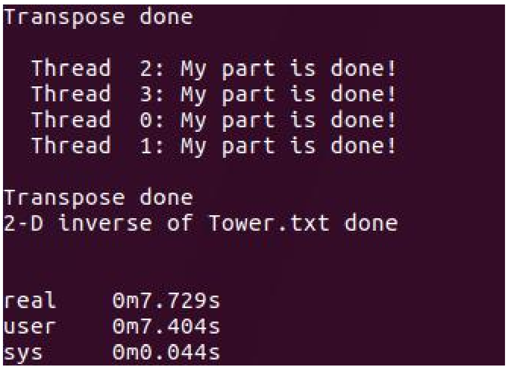
多线程 FFT 程序运行过程的截图如下：

FFT 程序的输出

1 个线程时：

2 个线程时：

4 个线程时：



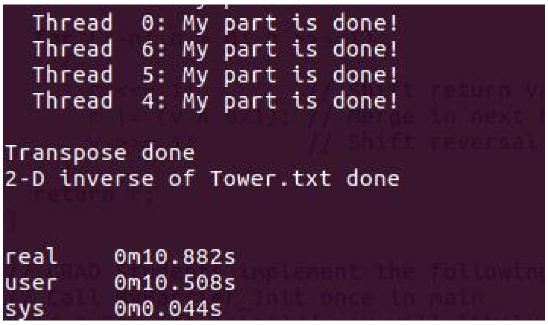
```
Transpose done

  Thread  2: My part is done!
  Thread  3: 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	0m7.729s
user	0m7.404s
sys	0m0.044s
```

8 个线程时:

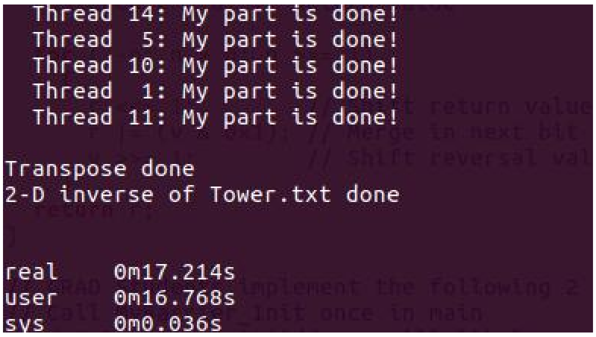


```
  Thread  0: My part is done!
  Thread  6: My part is done!
  Thread  5: My part is done!
  Thread  4: My part is done!

Transpose done
2-D inverse of Tower.txt done


real	0m10.882s
user	0m10.508s
sys	0m0.044s
```

16 个线程时：



```
  Thread 14: My part is done!
  Thread  5: My part is done!
  Thread 10: My part is done!
  Thread  1: My part is done!
  Thread 11: My part is done!

Transpose done
2-D inverse of Tower.txt done


real	0m17.214s
user	0m16.768s
sys	0m0.036s
```

在这个测试结果中，real 表示实际 FFT 程序运行时间，sys 表示内核运行时间，即线程运行时间，由此可得：

线程数	1	2	4	8	16
线程时间 ms	40	36	44	44	36
单线程时间 ms	40	18	11	5.5	2.25
加速比	1	2.22	3.64	7.27	17.78
理论加速比	1	2	4	8	16
误差	0%	11%	9%	9.1%	11.1%

分析和结论

从测试记录来看，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即多线程提升的性能，根据以上表格可知，理论上多线程的加速比应该是 1、2、4、8、16，但是实际上加速比却分别为 1、2.22、3.64、7.27、17.78，实际提升的性能与理论性能产生了误差，但保持在 10% 左右，这可能是由于实验误差，没有多次运行取其平均值。

在实验中，可以知道这些线程的总运行时间是相差不大的，都在 40ms 左右，但是根据测试结果，FFT 程序的运行时间在不断增大，可以知道，当线程增加时，线程运行时间没有改变，同时其他操作程序运行时间也没有改变，那么这些增加的时间来源于在进行线程间操作时的消耗：

1、首先是线程的创建与销毁时间，这个时间会随着线程的数量增加而增加；

2、然后时线程进行切换时花费的时间，每当线程进行切换时，即当线程阻塞或终止时，就要保存当前的线程信息，然后读取要切换的线程的信息，线程数越大，进行切换的次数越多，花费时间越多。