实验报告

实验名称（多线程FFT程序性能分析和测试）

物联1601 201601110208 方缙

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();

  // 初始化互斥锁和条件变量
  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
  clock_t start=clock();
  for(i=0; i < N_THREADS; ++i){
    pthread_create(&threads[i], 0, Transform2DTHread, (void *)i);
  }
  clock_t end=clock();
  printf("1 %d %d\n",N_THREADS,end-start);
  // 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
    start=clock();
    for(i=0; i < N_THREADS; ++i){
      pthread_create(&threads[i], 0, Transform2DTHread, (void *)i);
    }
    end=clock();
    printf("2 %d %d\n",N_THREADS,end-start);

    // 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

    start=clock();
    for(i=0; i < N_THREADS; ++i){
      pthread_create(&threads[i], 0, Transform2DTHread, (void *)i);
    }
    end=clock();
    printf("3 %d %d\n",N_THREADS,end-start);

    // 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

    start=clock();
    for(i=0; i < N_THREADS; ++i){
      pthread_create(&threads[i], 0, Transform2DTHread, (void *)i);
    }
    end=clock();
    printf("4 %d %d\n",N_THREADS,end-start);
    // 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，其中pthread是POSIX的线程标准，定义了创建和操纵线程的一套API。Pthreads定义了一套C语言的类型、函数与常量，它以pthread.h头文件和一个线程库实现。

Pthreads API中大致共有100个函数调用，全都以"pthread_"开头，并可以分为四类：

- 线程管理，例如创建线程，等待(join)线程，查询线程状态等。

- 互斥锁（Mutex）：创建、摧毁、锁定、解锁、设置属性等操作
- 条件变量（Condition Variable）：创建、摧毁、等待、通知、设置与查询属性等操作
- 使用了互斥锁的线程间的同步管理

pthread里面包含有如下的数据类型

- pthread_t：线程句柄.出于移植目的，不能把它作为整数处理，应使用函数pthread_equal()对两个线程ID进行比较。获取自身所在线程id使用函数pthread_self()。
- pthread_attr_t：线程属性。主要包括scope属性、detach属性、堆栈地址、堆栈大小、优先级。主要属性的意义如下：
    - __detachstate，表示新线程是否与进程中其他线程脱离同步。如果设置为PTHREAD_CREATE_DETACHED，则新线程不能用pthread_join()来同步，且在退出时自行释放所占用的资源。缺省为PTHREAD_CREATE_JOINABLE状态。可以在线程创建并运行以后用pthread_detach()来设置。一旦设置为PTHREAD_CREATE_DETACHED状态，不论是创建时设置还是运行时设置，则不能再恢复到PTHREAD_CREATE_JOINABLE状态。
    - __schedpolicy，表示新线程的调度策略，包括SCHED_OTHER（正常、非实时）、SCHED_RR（实时、轮转法）和SCHED_FIFO（实时、先入先出）三种，缺省为SCHED_OTHER，后两种调度策略仅对超级用户有效。运行时可以用过pthread_setschedparam()来改变。
    - __schedparam，一个struct sched_param结构，目前仅有一个sched_priority整型变量表示线程的运行优先级。这个参数仅当调度策略为实时（即SCHED_RR或SCHED_FIFO）时才有效，并可以在运行时通过pthread_setschedparam()函数来改变，缺省为0。系统支持的最大和最小的优先级值可以用函数sched_get_priority_max和sched_get_priority_min得到。
    - __inheritsched，有两种值可供选择：PTHREAD_EXPLICIT_SCHED和PTHREAD_INHERIT_SCHED，前者表示新线程使用显式指定调度策略和调度参数（即attr中的值），而后者表示继承调用者线程的值。缺省为PTHREAD_EXPLICIT_SCHED。
    - __scope，表示线程间竞争CPU的范围，也就是说线程优先级的有效范围。POSIX的标准中定义了两个值：PTHREAD_SCOPE_SYSTEM和PTHREAD_SCOPE_PROCESS，前者表示与系统中所有线程一起竞争CPU时间，后者表示仅与同进程中的线程竞争CPU。目前LinuxThreads仅实现了PTHREAD_SCOPE_SYSTEM一值。
- pthread_barrier_t：同步屏障数据类型
- pthread_mutex_t：mutex数据类型
- pthread_cond_t：条件变量数据类型

多线程FFT程序性能分析

通过分析多线程FFT程序代码，多线程改进部分占原来所需计算部分的50%，理想情况下升级加速比为N_THREADS,可以推断多线程FFT程序相对于单线程情况可达到的加速比应为：

$$\frac{2}{1+\frac{1}{NTHREADS}}$$

在该程序中，主要的变换过程在Transform2D()函数中，在该函数中首先初始化互斥量以及条件变量以便接下来的代码使用。然后程序总共有四次for循环创建N_THREADS个线程，然后四次嵌套for循环。线程中运行的函数主要是将row行的n个数据进行转化然后根据Danielson-Lanczos 算法转换。其中row由N_THREADS决定,N_THREADS越多，单个线程的row则越少。因为每个线程可以看成是并行的，所以单个线程运行函数的时间复杂度为

$$O(\frac{k*(n\log n+n)}{NTHREADS})$$

其中k是一个和输入规模n和线程数N_THREADS无关的常数。 在所有线程都运行完成之后，会有一个嵌套for循环进行矩阵转置，一共有4次。所以整个程序的时间复杂度为

$$O(\frac{k*(n\log n+n)}{NTHREADS}+a*n^2)$$

其中k与a都是与输入规模n和线程数N_THREADS无关的常量。

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 i7-6600U	
内存	DDR4 16GB	
操作系统	Ubuntu 16.04 LTS	虚拟机中运行

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	
线程数目	1,2,4,8,16	

多线程FFT程序运行过程的截图如下：

```
fj@ubuntu:~/Documents/Exp2/pthreads-fft2d-master$ ./threadDFT2d 4
1 4 81
  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 4 76
  Thread  2: My part is done!
  Thread  3: My part is done!
  Thread  1: My part is done!
  Thread  0: My part is done!

Transpose done
2-D transform of Tower.txt done

3 4 72
  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

4 4 105
  Thread  3: My part is done!
  Thread  2: My part is done!
  Thread  0: My part is done!
```

FFT程序的输出

线程数	时间1(ms)	时间2(ms)	时间3(ms)	时间4(ms)
1	41	50	39	94
2	61	62	45	57
4	81	76	72	105
8	133	138	224	229
16	255	343	346	301

分析和结论

由于采用多线程技术，所以理论上来说，矩阵转换部分的运行时间应该为单线程的1/N_THREADS倍。

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，考虑到我的处理器是双核四线程的，所以当线程数小于4时，程序有一定的速度提升。

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

在实际运行过程中进程间的上下文切换需要额外的时间。我的CPU本身只有双核4线程，所以当创建16个线程时并不会16个线程并行，而是会一部分线程在一个时间片并行，然后另一部分线程在下一个时间片并行。所以导致了真实的运行时间没有达到理想运行时间。

此外考虑到这个程序是一个cpu密集型的程序，主要耗时在于计算，所以线程切换时的缓存不命中也可能有一定的原因，通过linux平台下的valgrind测试4个线程的cache命中情况如下：

线程数	读不命中	写不命中	分支预测失败率
1	4,344,604	526,843	3.4%
4	4,368,517	527,359	3.0%
16	4,368,613	528,965	1.9%

可以看到，随着线程数的增加，cache读写的不命中数都有所增加。但是值得注意的是，随着线程数的增加，分支预测的失败率有所减少。所以如果能够处理好cache的竞争问题，应该能够使多线程程序的运行时间达到一个较为理想的值。