实验报告

实验名称（多线程 FFT 程序性能分析和测试）

班级 智能 1602 学号 201608010708 姓名田宗杭

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

多线程 FFT 的代码可以参考[这里](https://github.com/urgv/pthreads-fft2d)。

该代码采用了 pthread 库来实现多线程，其中...(此处请补充 pthread 的使用)

Pthread 的使用：
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。Windows 操作系统也有其移植版 pthreads-win32。

使用之前需要头文件：<pthread.h>

数据类型：
pthread_t：线程 ID
pthread_attr_t：线程属性

下面为 pthread 的 API：
pthread_t：线程 ID
pthread_attr_t：线程属性

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

与同步有关的：
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
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
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

多线程 FFT 程序性能分析

通过分析多线程 FFT 程序代码，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：

(此处请补充多线程 FFT 程序代码的性能分析)

与传统的并行算法相比，采用多线程技术的并行算法的一个特点是算法设计者无需考虑计

算机具体的硬件结构,如网络拓补结构、存储器层次结构等;另外,多线程技术本身就包含了一定的并行性,使用多线程技术来进行并行程序设计也就显得比较合理了.线程是一种特殊的对象,是操作系统执行多任务的一部分,它允许应用程序的一部分独立于其他对象而单独运行,因此也就脱离了应用程序常规的执行顺序.可以将进程分解为多个线程,以共享分配给它的时间片.单个处理器一次只能执行一个线程,进程中的每个线程都被分配了一部分执行时间,以完成它的工作.但是,如果计算机使用了多个处理器,那么,在同一时刻将会有多个线程被分配执行时间(此工作将由操作系统来完成),进程的各个独立的对象(线程)将并发地执行 .如果进程没有被分解,即使有多个处理器,那么进程一次也只能获得一个执行时间.因此,在多处理器平台上,使用多线程的应用程序将获得更高的性能.但是,由于各个线程是相互独立的对象,而且线程一旦开创后便很难得到控制,因此,在设计多线程算法时,应设法找出其算法中的并行性。

算法第一阶段所需的计算时间为 O(N/s+N),第一阶段的第一步所需计算时间为O(1),第二到三步所需的计算时间为O(N1bN/s-N1bα/s-N1bβ/(2s),而第五步的计算时间为O(N1bα/(2s))所以算法的总时间复杂度为O(N(21bN-1b(α β)/(2s))根据线程与处理核数之间的关系 s=2^r*p(r=0,1,……)可以得出算法的加速比：S(p)=O((21bN*2^r)/(21bN-1b(α β) *p)
算法理论上获得了近似线性加速。
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	DDR3 16GB	
操作系统	Ubuntu 18.04 LTS	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：

FFT 程序的输出

开一个线程进行处理时：

```
  Thread  0: My part is done!

1-D transform of Tower.txt done
it took 1.383806 seconds
Transpose done

  Thread  0: My part is done!
it took 0.679889 seconds

Transpose done
2-D transform of Tower.txt done

  Thread  0: My part is done!

Transpose done

it took 0.693364 seconds
  Thread  0: My part is done!

Transpose done
it took 0.725604 seconds
2-D inverse of Tower.txt done

Performance counter stats for './threadDFT2d':

      9905.335741      task-clock (msec)         #    1.479 CPUs utilized
               27      context-switches          #    0.003 K/sec
                1      cpu-migrations            #    0.000 K/sec
            4,211      page-faults               #    0.425 K/sec
  <not supported>      cycles
  <not supported>      instructions
  <not supported>      branches
  <not supported>      branch-misses

      6.695215259 seconds time elapsed
```

开两个线程进行处理时：

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

1-D transform of Tower.txt done
it took 1.292361 seconds
Transpose done

  Thread  0: My part is done!
  Thread  1: My part is done!
it took 0.311316 seconds

Transpose done
2-D transform of Tower.txt done

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

Transpose done

it took 0.313299 seconds
  Thread  0: My part is done!
  Thread  1: My part is done!

Transpose done
it took 0.337725 seconds
2-D inverse of Tower.txt done

 Performance counter stats for './threadDFT2d':

      10086.165431      task-clock (msec)         #    1.818 CPUs utilized
                45      context-switches          #    0.004 K/sec
                 1      cpu-migrations            #    0.000 K/sec
             4,223      page-faults               #    0.419 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

       5.548573225 seconds time elapsed
```

开四个线程进行处理时：

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

1-D transform of Tower.txt done
it took 1.419833 seconds
Transpose done

  Thread  2: My part is done!
  Thread  1: My part is done!
  Thread  0: My part is done!
  Thread  3: My part is done!
it took 0.190098 seconds

Transpose done
2-D transform of Tower.txt done

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

Transpose done

it took 0.211434 seconds
  Thread  1: My part is done!
  Thread  2: My part is done!
  Thread  3: My part is done!
  Thread  0: My part is done!

Transpose done
it took 0.197294 seconds
2-D inverse of Tower.txt done
```

```
 Performance counter stats for './threadDFT2d':

      12456.626545      task-clock (msec)         #    2.425 CPUs utilized
                92      context-switches          #    0.007 K/sec
                 5      cpu-migrations            #    0.000 K/sec
             4,239      page-faults               #    0.340 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

       5.136849562 seconds time elapsed
```

开八个线程进行处理时：

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

1-D transform of Tower.txt done
it took 2.044960 seconds
Transpose done

  Thread  6: My part is done!
  Thread  2: My part is done!
  Thread  3: My part is done!
  Thread  4: My part is done!
  Thread  0: My part is done!
  Thread  1: My part is done!
  Thread  5: My part is done!
  Thread  7: My part is done!
it took 0.205463 seconds

Transpose done
2-D transform of Tower.txt done

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

Transpose done

it took 0.235341 seconds
  Thread  4: My part is done!
  Thread  3: My part is done!
  Thread  0: My part is done!
  Thread  1: My part is done!
  Thread  6: My part is done!
  Thread  5: My part is done!
  Thread  7: My part is done!
  Thread  2: My part is done!

Transpose done
it took 0.230793 seconds
```

```
Performance counter stats for './threadDFT2d':

     23290.112458      task-clock (msec)         #    4.150 CPUs utilized
           1,017      context-switches          #    0.044 K/sec
              48      cpu-migrations            #    0.002 K/sec
           4,272      page-faults               #    0.183 K/sec
  <not supported>      cycles
  <not supported>      instructions
  <not supported>      branches
  <not supported>      branch-misses

      5.611935973 seconds time elapsed
```

开十六个线程进行处理时：

```
  Thread  0: My part is done!
  Thread  6: My part is done!
  Thread  9: My part is done!
  Thread  2: My part is done!
  Thread  5: My part is done!
  Thread 14: My part is done!
  Thread  1: My part is done!
  Thread 10: My part is done!
  Thread  3: My part is done!
  Thread 15: My part is done!
  Thread 11: My part is done!
  Thread  4: My part is done!
  Thread 12: My part is done!
  Thread  7: My part is done!
  Thread  8: My part is done!
  Thread 13: My part is done!

1-D transform of Tower.txt done
it took 3.791099 seconds
Transpose done

  Thread  0: My part is done!
  Thread  4: My part is done!
  Thread  2: My part is done!
  Thread  5: My part is done!
  Thread  6: My part is done!
  Thread  3: My part is done!
  Thread 10: My part is done!
  Thread 11: My part is done!
  Thread  7: My part is done!
  Thread  9: My part is done!
  Thread  8: My part is done!
  Thread  1: My part is done!
  Thread 15: My part is done!
  Thread 14: My part is done!
  Thread 12: My part is done!
  Thread 13: My part is done!
it took 0.276032 seconds

Transpose done
2-D transform of Tower.txt done
```

```
  Thread  6: My part is done!
  Thread  1: My part is done!
  Thread  0: My part is done!
  Thread  5: My part is done!
  Thread 14: My part is done!
  Thread 13: My part is done!
  Thread  2: My part is done!
  Thread  4: My part is done!
  Thread 12: My part is done!
  Thread  8: My part is done!
  Thread 11: My part is done!
  Thread  9: My part is done!
  Thread  7: My part is done!
  Thread 10: My part is done!
  Thread 15: My part is done!
  Thread  3: My part is done!

Transpose done

it took 0.281377 seconds
  Thread  0: My part is done!
  Thread  4: My part is done!
  Thread  5: My part is done!
  Thread  8: My part is done!
  Thread 13: My part is done!
  Thread 12: My part is done!
  Thread  3: My part is done!
  Thread  2: My part is done!
  Thread  1: My part is done!
  Thread 14: My part is done!
  Thread  6: My part is done!
  Thread 10: My part is done!
  Thread 15: My part is done!
  Thread  7: My part is done!
  Thread 11: My part is done!
  Thread  9: My part is done!

Transpose done
it took 0.270114 seconds
2-D inverse of Tower.txt done
```

```
 Performance counter stats for './threadDFT2d':

      40380.675119      task-clock (msec)         #    5.498 CPUs utilized
             3,586      context-switches          #    0.089 K/sec
               131      cpu-migrations            #    0.003 K/sec
             4,338      page-faults               #    0.107 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

       7.343986066 seconds time elapsed
```

开三十二个线程进行处理时：

```
  Thread  0: My part is done!
  Thread  1: My part is done!
  Thread 24: My part is done!
  Thread  9: My part is done!
  Thread  8: My part is done!
  Thread  2: My part is done!
  Thread 21: My part is done!
  Thread  4: My part is done!
  Thread 20: My part is done!
  Thread 15: My part is done!
  Thread  7: My part is done!
  Thread 16: My part is done!
  Thread 23: My part is done!
  Thread 29: My part is done!
  Thread 19: My part is done!
  Thread 22: My part is done!
  Thread 10: My part is done!
  Thread  3: My part is done!
  Thread  6: My part is done!
  Thread 14: My part is done!
  Thread 12: My part is done!
  Thread 28: My part is done!
  Thread 11: My part is done!
  Thread 17: My part is done!
  Thread 31: My part is done!
  Thread  5: My part is done!
  Thread 18: My part is done!
  Thread 13: My part is done!
  Thread 27: My part is done!
  Thread 25: My part is done!
  Thread 26: My part is done!
  Thread 30: My part is done!

1-D transform of Tower.txt done
it took 7.491690 seconds
Transpose done
```

```
  Thread  1: My part is done!
  Thread  0: My part is done!
  Thread  3: My part is done!
  Thread 10: My part is done!
  Thread  4: My part is done!
  Thread 22: My part is done!
  Thread  2: My part is done!
  Thread  5: My part is done!
  Thread 23: My part is done!
  Thread 30: My part is done!
  Thread  9: My part is done!
  Thread 17: My part is done!
  Thread 20: My part is done!
  Thread 28: My part is done!
  Thread 12: My part is done!
  Thread 21: My part is done!
  Thread 31: My part is done!
  Thread 14: My part is done!
  Thread 29: My part is done!
  Thread 18: My part is done!
  Thread 16: My part is done!
  Thread  6: My part is done!
  Thread 27: My part is done!
  Thread  8: My part is done!
  Thread 19: My part is done!
  Thread 24: My part is done!
  Thread 13: My part is done!
  Thread  7: My part is done!
  Thread 11: My part is done!
  Thread 26: My part is done!
  Thread 15: My part is done!
  Thread 25: My part is done!
it took 0.240883 seconds

Transpose done
2-D transform of Tower.txt done
```

```
  Thread  3: My part is done!
  Thread  0: My part is done!
  Thread  9: My part is done!
  Thread  5: My part is done!
  Thread  1: My part is done!
  Thread  7: My part is done!
  Thread 11: My part is done!
  Thread 10: My part is done!
  Thread 15: My part is done!
  Thread 14: My part is done!
  Thread 18: My part is done!
  Thread  6: My part is done!
  Thread 31: My part is done!
  Thread 24: My part is done!
  Thread 30: My part is done!
  Thread 19: My part is done!
  Thread  2: My part is done!
  Thread 22: My part is done!
  Thread 17: My part is done!
  Thread 20: My part is done!
  Thread  8: My part is done!
  Thread 29: My part is done!
  Thread 28: My part is done!
  Thread  4: My part is done!
  Thread 23: My part is done!
  Thread 26: My part is done!
  Thread 16: My part is done!
  Thread 13: My part is done!
  Thread 12: My part is done!
  Thread 25: My part is done!
  Thread 27: My part is done!
  Thread 21: My part is done!

Transpose done
Performance counter stats for './threadDFT2d':

     63799.417928      task-clock (msec)         #    6.063 CPUs utilized
           8,581      context-switches          #    0.134 K/sec
             185      cpu-migrations            #    0.003 K/sec
           4,465      page-faults               #    0.070 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

     10.522983586 seconds time elapsed
```

线程数	总的时间	上下文切换	CPU 的丢失	线程部分的运行时间	加速比
1	6.69	27	1	0.7	1
2	5.55	45	1	0.32	2.1875
4	5.14	92	5	0.20	3.5
8	5.61	1017	48	0.22	3.181
16	7.34	3586	131	0.275	2.55
32	10.52	8581	185	0.28	2.5

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先减少后又增大，当线程数为 4 时（计算机为双核)，加速效果达到最大，执行时间最少，上下文切换随着线程数量的增加也逐渐增加，同时 CPU 的丢失也逐渐增加。其相对于单线程情况的加速比分别为当线程数少于计算机所能并行运行的最大线程数时，随着线程的数量的增加，加速比近似为线性，当线程数大于计算机所能并行运行的最大线程数时，随着线程数增加，加速比逐渐减少。多线程的开销主要为上下文切换以及线程创建的开销。

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？
多线程相对于单线程提升了性能，当线程数少于计算机所能并行运行的最大线程数时与理论预测近似相等，但当线程数大于并行运行的最大线程数时与理论预测相差较大，主要是由于线程越多，线程的创建开销越大，且线程越多，线程处于阻塞的时间越长，上下文的切换开销也增大。