实验报告

实验名称（多线程 FFT 程序性能分析和测试）

201608010627 智能 1602 任小禹

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

实验内容

多线程 FFT 代码

多线程 FFT 的代码。

该代码采用了 pthread 库来实现多线程，其中

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。

pthread_t :线程 ID

pthread_attr_t :线程属性

pthread_mutex_t 互斥锁

pthread_cond_t: 条件变量的设定

pthread_create()：创建一个线程

pthread_attr_init() :初始化线程的属性

pthread_attr_destroy() :删除线程的属性

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

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：N（N 为线程数）

根据阿姆达尔定律：

S=1/(1-a+a/n)，其中，a 为并行计算部分所占比例，n 为并行处理结点个数。

根据我们的代码主函数：

```
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.
}
```

关键部分是：Tranform2D 转化函数，我们在这里创建多个线程。1-a->0 时，s->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 i7-8550U	
内存	DDR4 8GB	
操作系统	Ubuntu 18.04 LTS	

测试记录

多线程 FFT 程序的测试参数如下：

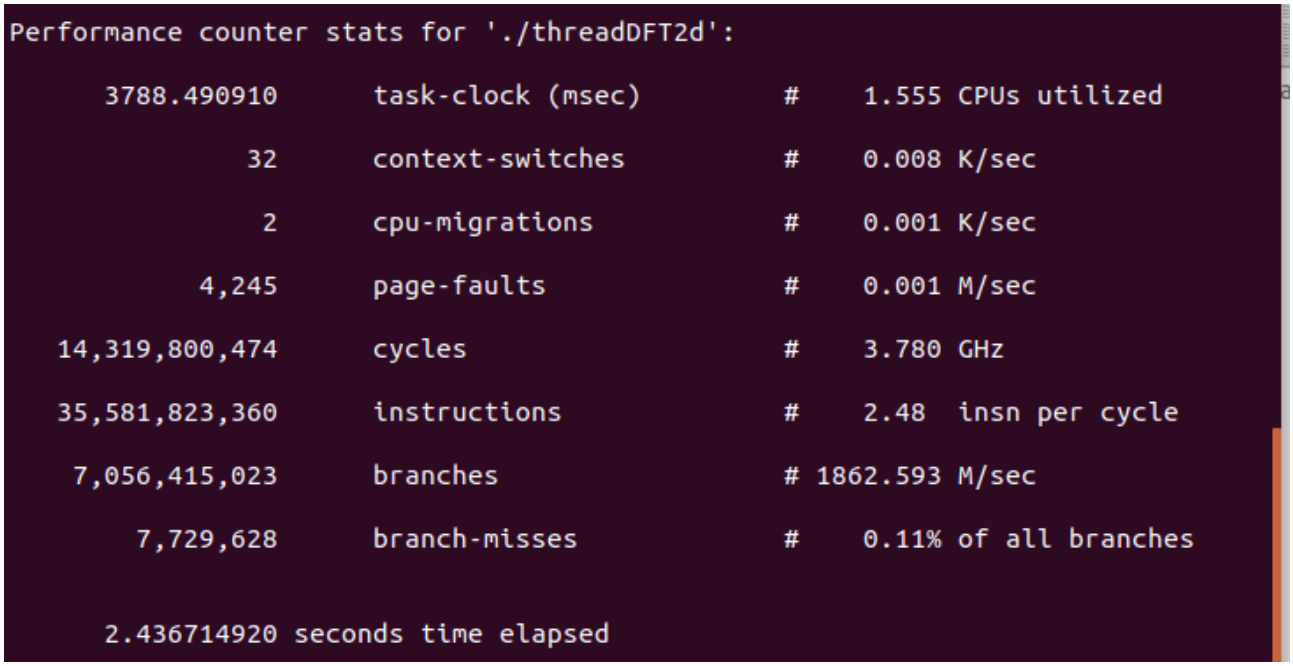
参数	取值	备注
数据规模	1024	
线程数目	1,2,4,8,16,32	

多线程 FFT 程序运行过程的截图如下：

FFT 程序的输出

使用 perf 工具对程序的时间进行追踪：

1 thread



2 threads

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

      3970.671726      task-clock (msec)         #    1.939 CPUs utilized
               43      context-switches          #    0.011 K/sec
                4      cpu-migrations            #    0.001 K/sec
            4,254      page-faults               #    0.001 M/sec
   14,609,783,306      cycles                    #    3.679 GHz
   36,738,814,884      instructions              #    2.51  insn per cycle
    7,342,488,641      branches                  # 1849.180 M/sec
        7,803,144      branch-misses             #    0.11% of all branches

      2.047718691 seconds time elapsed
```

4 threads

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

      5744.246121      task-clock (msec)         #    2.841 CPUs utilized
               59      context-switches          #    0.010 K/sec
                0      cpu-migrations            #    0.000 K/sec
            4,272      page-faults               #    0.744 K/sec
   19,842,244,433      cycles                    #    3.454 GHz
   46,403,741,105      instructions              #    2.34  insn per cycle
    9,758,551,753      branches                  # 1698.839 M/sec
        7,809,049      branch-misses             #    0.08% of all branches

      2.021960535 seconds time elapsed
```

8 threads

16 threads

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

     18563.395183      task-clock (msec)         #    5.575 CPUs utilized
            2,720      context-switches          #    0.147 K/sec
               77      cpu-migrations            #    0.004 K/sec
            4,376      page-faults               #    0.236 K/sec
   61,564,175,502      cycles                    #    3.316 GHz
  115,992,931,151      instructions              #    1.88  insn per cycle
   27,156,978,626      branches                  # 1462.932 M/sec
        8,380,017      branch-misses             #    0.03% of all branches

      3.329624429 seconds time elapsed
```

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

     15098.268494      task-clock (msec)         #    5.182 CPUs utilized
            1,812      context-switches          #    0.120 K/sec
               29      cpu-migrations            #    0.002 K/sec
            4,309      page-faults               #    0.285 K/sec
   49,947,575,707      cycles                    #    3.308 GHz
   92,489,139,085      instructions              #    1.85  insn per cycle
   21,281,366,008      branches                  # 1409.524 M/sec
        8,332,226      branch-misses             #    0.04% of all branches

      2.913862465 seconds time elapsed
```

32 threads

分析和结论

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

     33047.742327      task-clock (msec)         #    6.352 CPUs utilized
            6,841      context-switches          #    0.207 K/sec
              203      cpu-migrations            #    0.006 K/sec
            4,510      page-faults               #    0.136 K/sec
  109,499,056,883      cycles                    #    3.313 GHz
  211,787,893,193      instructions              #    1.93  insn per cycle
   51,103,464,085      branches                  # 1546.353 M/sec
        8,867,026      branch-misses             #    0.02% of all branches

      5.202492829 seconds time elapsed
```

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，先减小后增大，其相对于单线程情况的加速比如图：

线程数	运行时间（s)	理论加速比	加速比
1	2.44	1	1
2	2.05	2	2.38
4	2.02	4	4.83
8	2.91	8	6.71
16	3.33	16	11.72
32	5.20	32	15.02

思考题

1. pthread 是什么？怎么使用?

   POSIX 线程是 POSIX 的线程标准，定义了创建和操纵线程的一套 API，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

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?

理论上 N 个线程是单线程性能的 N 倍，多线程的开销有线程的创建和线程上下文切换，线程越多，其这两类开销越多。

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？

   我们可以看到，多线程相对于单线程的性能先提升后减弱，在我的测试环境下，于 4 线程达到最大的加速比，以后性能逐渐减弱，8 线程与理论值相差不多，32 线程的加速比比理论值一半还低，其性能甚至低于单线程的性能。分析可知，首先的原因是多线程有一定的开销，线程的创建和上下文切换都需要一定时间，而且随着线程数增多，其开销也越来越大。还有机器环境的问题，我的机器是四核八线程的，所以线程数过多时，处理器也无法立即处理，这也就增加了运行时间。所以线程数很多时反倒性能不如单线程。