实验报告

实验名称（多线程 FFT 程序性能分析和测试）

智能 1602 201608010609 李鹏飞

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)。

Pthread 介绍

pthread 是一个线程库，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。
在示例代码中采用了 pthread 库来实现多线程：

其中各个变量如下：

pthread_mutex_t:互斥变量使用的特定的数据类型
pthread_cond_t:条件变量的设定

其中各个函数含义如下：

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_destroy()：删除线程的属性
pthread_kill()：向线程发送一个信号

多线程 FFT 程序性能分析

在这里我们利用阿姆达尔定律进行分析：

对于固定负载情况下描述并行处理效果的加速比 s，公式如下：

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。这样，当 1-a=0 时，(即没有串行，只有并行)最大加速比 s=n；当 a=0 时（即只有串行，没有并行），最小加速比 s=1；当 n→∞时，极限加速比 s→ 1/（1-a），这也就是加速比的上限。例如，若串行代码占整个代码的 25%，则并行处理的总体性能不可能超过 4。

在这里我们发现在多线程 FFT 算法之中主要优化的部分为 Transform2D 函数，在这个函数之中创建相应的线程实现并行操作。所以根据阿姆达尔定律我们可以求得理论上加速比应该为 N，N 为创建的线程数。

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 i5-6700U	
内存	DDR4 12GB	
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：
在这里我们利用 perf 调用工具去进行跟踪。求得整个 FFT 运行时间，从宏观上观察分析多线程对于程序运行时间的影响。

线程个数 1：

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

      5287.230517      task-clock (msec)         #    1.553 CPUs utilized
               66      context-switches          #    0.012 K/sec
                1      cpu-migrations            #    0.000 K/sec
            4,245      page-faults               #    0.803 K/sec
   14,301,493,250      cycles                    #    2.705 GHz
   35,681,613,830      instructions              #    2.49  insn per cycle
    7,074,900,987      branches                  # 1338.111 M/sec
        7,967,463      branch-misses             #    0.11% of all branches

      3.404416135 seconds time elapsed
```

线程个数 2：

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

      5424.838474      task-clock (msec)         #    1.912 CPUs utilized
               96      context-switches          #    0.018 K/sec
                3      cpu-migrations            #    0.001 K/sec
            4,255      page-faults               #    0.784 K/sec
   14,596,583,155      cycles                    #    2.691 GHz
   36,776,955,610      instructions              #    2.52  insn per cycle
    7,351,970,487      branches                  # 1355.242 M/sec
        7,714,319      branch-misses             #    0.10% of all branches

      2.836736238 seconds time elapsed
```

线程个数 4：

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

      7472.309303      task-clock (msec)         #    2.393 CPUs utilized
            2,104      context-switches          #    0.282 K/sec
               25      cpu-migrations            #    0.003 K/sec
            4,272      page-faults               #    0.572 K/sec
   20,483,090,840      cycles                    #    2.741 GHz
   60,023,926,066      instructions              #    2.93  insn per cycle
   13,163,438,333      branches                  # 1761.629 M/sec
        8,297,691      branch-misses             #    0.06% of all branches

      3.122703808 seconds time elapsed
```

线程个数 8：

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

      8877.812763      task-clock (msec)         #    2.607 CPUs utilized
            1,258      context-switches          #    0.142 K/sec
               36      cpu-migrations            #    0.004 K/sec
            4,303      page-faults               #    0.485 K/sec
   24,424,039,114      cycles                    #    2.751 GHz
   75,680,536,866      instructions              #    3.10  insn per cycle
   17,077,131,406      branches                  # 1923.574 M/sec
       10,176,926      branch-misses             #    0.06% of all branches

      3.405752972 seconds time elapsed
```

线程个数 16：

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

     13944.261878      task-clock (msec)         #    2.974 CPUs utilized
            2,802      context-switches          #    0.201 K/sec
               83      cpu-migrations            #    0.006 K/sec
            4,374      page-faults               #    0.314 K/sec
   38,748,068,948      cycles                    #    2.779 GHz
  132,734,230,116      instructions              #    3.43  insn per cycle
   31,343,098,737      branches                  # 2247.742 M/sec
        7,905,313      branch-misses             #    0.03% of all branches

      4.688094161 seconds time elapsed
```

从上面的运行结果中我们可以提取到如下信息：

线程数	时间（s）	上下文切换（次）	丢失率	加速比（单线程/多线程中的每个线程）	理论值
1	3.40	66	0.11%	1	1
2	2.83	96	0.10%	2.25	2
4	3.12	2104	0.06%	4.25	4
8	3.40	1258	0.06%	7.55	8
16	4.68	2802	0.03%	11.33	16

分析、结论与思考题

1.从上述表格的分析结果来看，随着线程数目的增加，时间先减小后增大。上下文的切换次数增多，分支预测的丢失率降低，加速比低于预估值。当线程数为 1 和为 2 的时候加速比大于预估值，是因为我们在这里利用 perf 追踪工具，从宏观上查看其时间，所以其变量创建的时间也会包含在其中，这就使得多线程的加速比可能会大于理论值。

2.思考题：理论上提升多线程相对于单线程提升 N 倍(N 为多线程线程数目)
但是多线程的开销有线程的创建和上下文切换.都属于只能减小,不可避免的开销.所以在上述的结果之中体现为随着线程数目的增加，上下文切换次数多，开销增大，加速比就会减小。

3.思考题：合适的多线程可以提升程序的性能,但是如果线程数目设置的不合理,将会导致程序的运行缓慢可能的原因:
(1)计算机本身内核数目不够,假设计算机为 4 核,如果要运行一个 16 线程的程序。就会导致其他线程需要等待,这就使得计算机无法发挥多线程的优势
(2)因为数据之间可能有数据纠缠关系,这就导致了线程之间的切换可能会有间隔。所以当线程数目过多时,可能效率反而会下降.
(3)在程序执行过程之中，如果一个程序发生了阻塞，可能会导致连锁反应，影响整个程序的运行时间，这也可能会造成程序运行的缓慢。