多线程 FFT 程序性能分析和测试

智能 1602 孟祥炜 201607020301

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。

实验内容

Pthread 介绍

实验中使用使用 pthread 实现多线程，这里对 pthread 进行一个简单的介绍

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。

常用的操作函数

```
#include <pthread.h>
//新建线程
int pthread_create(pthread_t *restrict tidp, constpthread_attr_t *restrict attr,void*(*start_rtn)(void*),void *restrict arg);

//线程终止
void pthread_exit(void *rval_ptr);//线程自身主动退出
int pthread_join(pthread_t tid, void **rval_ptr);//其他线程阻塞自身，等待 tid 退出

//线程清理
voidpthread_cleanup_push(void(*rtn)(void*), void *arg);
voidpthread_cleanup_pop(intexecute);
```

本实验代码中出现的与 pthread 相关的变量和函数

变量

pthread_mutex_t 互斥锁

pthread_cond_t 条件变量

函数

pthread_mutex_init() 初始化互斥锁

pthread_cond_init()：初始化条件变量

pthread_mutex_lock 锁定互斥锁

pthread_mutex_unlock 解锁互斥锁

ppthread_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 程序性能理论分析

通过对多线程 FFT 程序代码的分析，我们可以看到，该程序的多个线程需要向同一个文件写入输出数据，除此以外没有不存在一个线程的运行状态影响另一个线程运行状态的情况。但是这种影响可以基本忽略，因为各个线程基本不可能同时完成计算并需要写数据，所以不存在同时需要执行写操作而造成的空闲等待时间误差。

所以可以认为各个线程是原任务的完全相等的子问题。在进行 n 线程并发的情况下，理论运算时间应该是单线程的$\frac{1}{n}$，也就是说理论加速比=线程并发数 n。

时间测量

由于使用 clock_gettime()方法获得的时间不能直观的得到与线程时间有关的信息，我采用的是 time 命令获得运行时间信息

Time 命令的好处是，他的计时可以分别显示程序运行运行总时间 real、在用户态的时间 user、内核态的时间 sys

缺点是只能显示到毫秒级别

由于我们需要检测多线程是否可以提升性能，所以我们只需要观察 sys 时间

之所以没有选择 clock_gettime()方法，是由于，使用 clock_gettime()方法时，需要从线程创建开始计时，直到线程结束并销毁结束。但是这里面混杂了线程提交请求、在就绪队列中等待等非计算时间，而这是我们不希望的。另一方面，线程创建基本是同一时刻完成，也就是说计时起点基本相同，而只有当线程结束并销毁的时候计时结束，而由于大部分个人笔记本达不到 16 线程的并发数，所以我们记录的时间将包括非计算时间，从而使得所显示的时间比实际时间长，给我们一种线程越多时间越长的错觉。

举个例子，假设我们 CPU 支持最大 2 线程并发，现在我们有 4 线程需要并发。假设极端情况 CPU 从一个线程创建开始就完全执行当前程序内的线程（而不执行其他应用及系统的线程），那么理论情况如下（0 时刻开始计时）

假设创了四个线程：a1,a2,a3,a4			
计时时间	0	t1	t2
线程创建	CPU上运行的线程	CPU上运行的线程	
a1	a1	计时结束	
a2	a2	计时结束	
a3	由于最大支持2线程并发，所以只能等待	a3	计时结束
a4	由于最大支持2线程并发，所以只能等待	a4	计时结束
统计结果			
	运行时间		
a1	t1		
a2	t1		
a3	t2		
a4	t2		

（在上面的例子中，理论上 t2=2*t1）

所测得的 a3，a4 线程运行时间为 t2、t2，但是实际上 a3、a4 线程的运行时间应该是(t2-t1)、(t2-t1)

在这个例子中，如果支持 4 线程并发，那么最终显示的时间就都是 t1，与实际情况相符。

大部分个人笔记本最多支持 4 线程并发，或 8 线程并发，而所需要测得数据有 16 线程并发，因此就会产生上面所举例的问题。

另外在实际情况中，会有大量的其他应用进程（线程）和系统进程（线程）也在运行，如果使用上面的方法，同样也会把这些不是我们希望的时间计算进去，产生较大误差。

而 time 命令支持显示程序执行时的总时间 real、用户态时间 user、内核态时间 sys。我们在这里只需要使用内核态时间 sys。使用 Time 指令，当程序进入内核态时，sys 开始计时，而当进入非内核态时停止计时。由于这个程序里的所分的线程任务相同，所以我们用得到的 sys 时间/线程数，就可以得到一个线程的运行时间，也就是在支持当前线程并发数的机器上的运行时间。

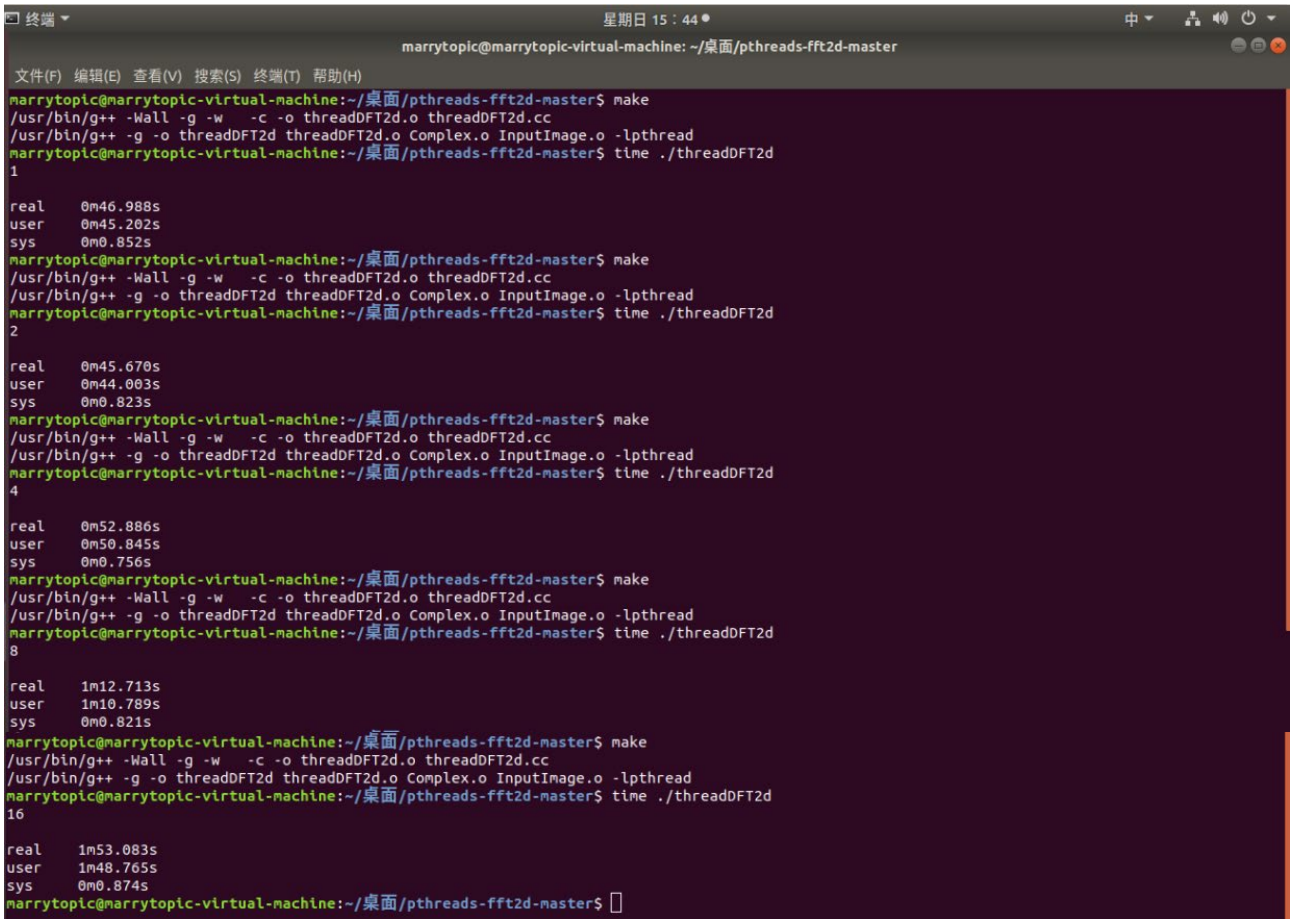
由于 time 命令只计算当前程序的各项时间，所以避免了其他应用进程（线程）和系统进程（线程）的影响。同时，由于使用了专用的计时器，当某个线程在等待时，由于该线程并不实际运行，所以不属于处于内核态，计时器也不会计时，只会在实际运行了该线程时才会计时。各个线程之间不共享计时器，因此不存在线程重叠时计数误差。

测试平台

CPU	Intel corei7-6500U（虚拟机中只分配了一个核）
内存	DDR3L 1.5GB
操作系统	Ubuntu 18.04 LTS

测试记录

使用 time 命令对程序进行计时的结果



终端输入 Time ./threadDFT2d 运行，显示当前的线程数和计时结果（10 次计时结果）

由于偶然误差的存在以及 time 命令的结果精确度较低，以上是 10 次计时结果之和（理论上应该取更多的次数，使得结果更稳定，但之所以没有选取更大的是因为，10 次的情况下程序运行时间已经达到 1 分钟左右，故没有选取更大的值）

提取出每个线程数对应的时间/10，得到每次的运行时间

线程数	1	2	4	8	16
运行时间(ms)	85.2	82.3	75.6	82.1	87.4

平均值 82.52，标准差 3.98，误差范围-8.4%~+5.9，可以认为五种情况下的理论运行时间相同。而这也符合理论推导，理论上一个任务分成 n 份，每份完成原任务的$\frac{1}{n}$，最后所需的时间是 n*$\frac{1}{n}$=1，还是原任务的数量。

按照实验数据计算理论运行时间

对于每组数据，因为分割的每个线程任务完全相同，所以计算理论时间=运行时间/线程数，也就是在机器能够达到最大并发数时的运行时间。

线程数	1	2	4	8	16
运行时间/线程数 (ms)	85.2	41.15	18.9	10.2625	5.4625

使用实验数据计算加速比

线程数	2	4	8	16
实际加速比(与单线程相比)	2.07	4.51	8.30	15.60
理论加速比(与单线程相比)	2	4	8	16
误差	3.5%	12.75%	%3.75	2.5%

可以看到，加速比基本和并发线程数相等

分析和结论

从测试的过程和结果可以看到，FFT 程序执行时间随着线程数的增加而减少，相对于单线程的加速比分别为

线程数	2	4	8	16
实际加速比(与单线程相比)	2.07	4.51	8.30	15.60

根据实验结果，可以认为加速比等于并发线程数。

思考题：

1. pthread 是什么？怎么使用？
   实验内容中已经说明。
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？
   理论理论提升的性能=多线程数
   多线程的开销主要是线程的创建与销毁，但是比进程的开销小得多。
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？
   多线程相对单线程提升了性能
   与理论预测误差在 5%以内。
   可能的原因：线程创建、销毁、切换的开销，每次文件读写的时间的不同，偶然误差等。