天津大学
Tianjin University

并行计算实验报告

课程： 并行计算

实验题目： 使用 Pthread 实现荒野求生

姓名： 王淑军

学号： 2018216134

时间： 2018 年 11 月 14 日

目录

一、题目描述

使用 pthread 实现荒野求生，并进行性能分析。

二、实验内容与多线程介绍

2.1 实验内容

荒野求生：

在一个 1600*900 的空间内有若干个探险小队，每个探险小队有初始的位置和速度，速度的方向有八个(U，D，L，R，LU，LD，RU，RD)，探险小队若碰撞到空间边缘则会转弯，转弯规则为（U->RD,D->LU,L->RU,R->LD,LU->R,LD->U,RU->D,RD->L）,速度大小不变。若在某个时刻，有多个小队同时到达某一位置，则会发生冲突，冲突后速度最慢小队会生存下来，若最慢的小队不只一个，则所有此位置的小队全部同归于尽。

输入数据包含若干行，第一行为一个整数 T，表示结束时间，单位为 s。其余每一行表示一个小队在 0s 时的状态。前两列为队伍的 x 坐

标和 y 坐标。左下角为 0 坐标，向上为 y 坐标，向右为 x 坐标。第三列表示方向。第四列表示速度，为非负整数，单位为（格/s)。输出为若干时间后存活的小队的位置和速度大小以及方向。

注：仅考虑整数秒时的冲突。队伍的转弯和冲突是瞬时的，不消耗时间。

2.2 多线程介绍

如果不能从根本上更新当前 CPU 的架构(在很长一段时间内还不太可能)，那么继续提高 CPU 性能的方法就是超线程 CPU 模式。那么，作业系统、应用程序要发挥 CPU 的最大性能，就是要改变到以多线程编程模型为主的并行处理系统和并发式应用程序。

2.2.1 多线程出现的原因

多线程编程的目的，就是"最大限度地利用 CPU 资源"，当某一线程的处理不需要占用 CPU 而只和 I/O,OEMBIOS 等资源打交道时，让需要占用 CPU 资源的其它线程有机会获得 CPU 资源。从根本上说，这就是多线程编程的最终目的。

2.2.2 多线程的优缺点

优点：

使用线程可以把占据时间长的程序中的任务放到后台去处理

用户界面更加吸引人,这样比如用户点击了一个按钮去触发某件事件的处理,可以弹出一个进度条来显示处理的进度

程序的运行效率可能会提高

在一些等待的任务实现上如用户输入,文件读取和网络收发数据等,线程就比较有用了

缺点:

如果有大量的线程,会影响性能,因为操作系统需要在它们之间切换.

更多的线程需要更多的内存空间

线程中止需要考虑对程序运行的影响.

通常块模型数据是在多个线程间共享的,需要防止线程死锁情况的发生

2.2.3 多线程提高执行效率

即采用多线程不会提高程序的执行速度，反而会降低速度，但是对于用户来说,可以减少用户的响应时间。这个结论只是针对单 CPU,如果对于多 CPU 或者 CPU 采用超线程技术的话，采用多线程技术还是会提高程序的执行速度的。因为单线程只会映射到一个 CPU 上，而多线程会映射到多个 CPU 上，超线程技术本质是多线程硬件化，所以也会加快程序的执行速度。

三、研究过程

本节将会指出在实验的过程中尝试过的一些思路并给出为什么放弃这些思路的分析。

- 首先想到的是只将 run(it)从代码中提取出来，并行执行。但是这种思路过于简单。并行化程度太低，因此被放弃。
- 第二种思路是基于数据进行划分，分别将 list<Piece>或者 state 进行分割，我测试了基于这种思路的三种分割方式。

  1. 只将 list 进行分割成多个子 list 并行执行
  2. 只将 State 进行分割成多个子 state 并行执行
  3. 将 list 和 state 都进行分割

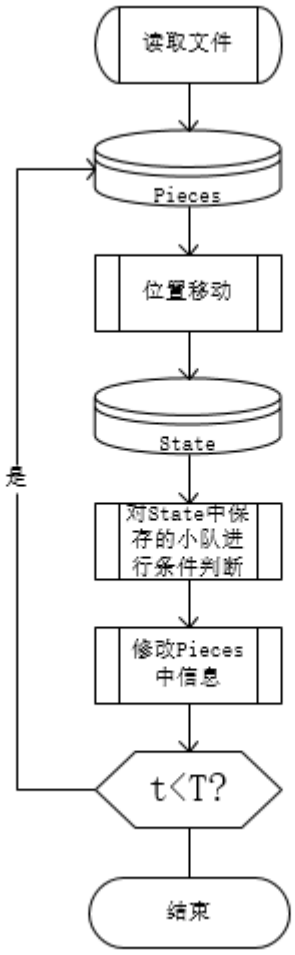
  在基于数据进行并行任务的三种划分中，我发现分割方式 1 和分割方式 3，无论使用几个线程，程序执行时间都不能小于串行程序。甚至会表现出随线程数量增多，执行时间更多的情况。

  发现实际的问题在于每一个子 list 执行的时间都大于串行代码中整个 list 的执行时间，经过测试发现是互斥锁存在问题。导致多线程绝大多数部分不能并行执行，只能串行执行。因此会表现出随着线程增多，并行程序执行时间变长的特征。

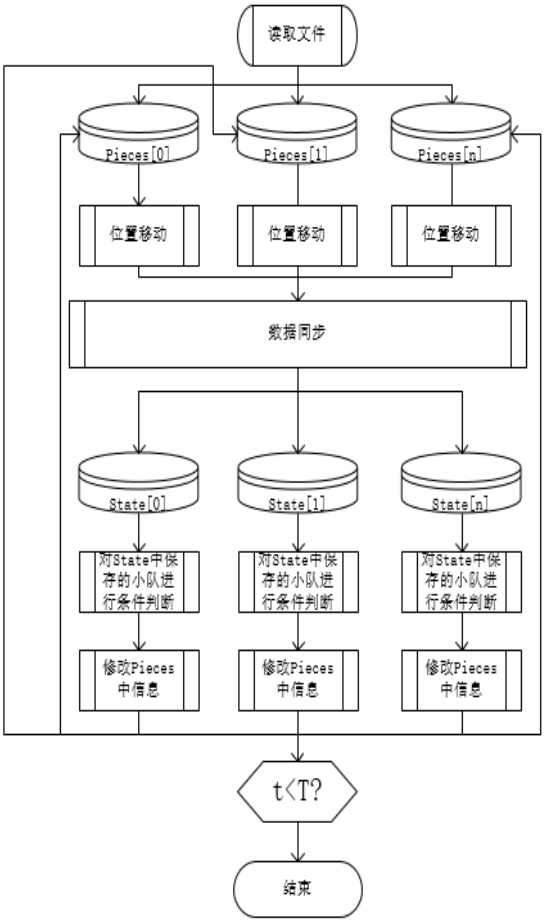
  经过试验发现基于数据的划分方法中，只有方法 2 能做到一定程度上提高串行程序的执行效率。在使用 8 个线程的时候，效率高于串行程序。
- 最后想到的也是最终采用的方法是按照小队所属区域进行划分。按照一个小队的 x 坐标和 y 坐标，并将 x,y 两个坐标映射到一维向量中，对所有小组进行线程分配，经过实际的测试验证发现，该方法的效果更加好。

四、程序流程图

串行代码流程图：



并行代码流程图：



五、伪代码以及具体实现

5.1 伪代码

```
begin
    for each line in ("input.txt")
```

```
    do
        loc <- getLoc(x,y) //将二维坐标映射为一维坐标，并按照线程数量进行区域划分
        subPieces[loc].push_back(data)
    end for

    for i <- 0 to NUM_THREADS
    do
        thread_create(thread[i],NULL,findShare,&i)
      //对所有小队进行移动，移动后位置不再属于原区域的小队，加入共享变量区域
    end for

    for i <- 0 to NUM_THREADS
    do
        thread_join(thread[i])
    end for

    for it <- in share
    do
        loc <- getLoc(it->x,it->y)
        put data in real thread
    end for

    for i <- 0 to NUM_THREADS
    do
        thread_create(&stateThreads[i],NULL,dealPiece) //对每一个 subPieces 中的小队按规定进行消除
    end for

    for i <- 0 to NUM_THREADS
    do
        thread_join[stateThread[i]]
    end for
end
```

5.2 关键代码

5.2.1 main 函数

```
int main(){
    ifstream input("input.txt");
    int tmpx,tmpy,tmps,T;
    string str;
```

```
    input >> T;
    while(input >> tmpx){
        input >> tmpy >> str >> tmps;
        int pos=getPostion(tmpx,tmpy);
        tempPieces[pos].emplace_back(tmpx,tmpy,str,tmps);
    }
    input.close();
    int index[NUM_THREADS];
    pthread_t threads[NUM_THREADS];
    pthread_t stateThreads[NUM_THREADS];
    int t=0;
    int i;
    int rc;
    while(t<T){
        ++t;
        pthread_mutex_init(&mutex,NULL);
        for(i=0;i<NUM_THREADS;++i)
        {
            index[i] = i;
            int ret=pthread_create(&threads[i],NULL,runPiece,(void *)&(index[i]));
            if(ret){
                cout<<"Error:unable to creat thread"<<ret<<endl;
                exit(-1);
            }
        }
        for(i=0;i<NUM_THREADS;++i){
            pthread_join(threads[i],NULL);
        }
        for(auto it=share_mem.begin();it!=share_mem.end();it++)
        {
            int pos=getPostion(it->x,it->y);
            tempPieces[pos].insert(tempPieces[pos].begin(),*it);
        }
         for(i=0;i<NUM_THREADS;i++)
        {
            index[i]=i;
            rc = pthread_create(&stateThreads[i], NULL,doSplit, (void
*)&(index[i]));
            if (rc){
                cout << "Error:unable to create thread," << rc << endl;
                exit(-1);
            }

        }
```

```
            for(i=0;i<NUM_THREADS;i++)
            {
                pthread_join(stateThreads[i],NULL);
            }
            int sum=0;
            for(i=0;i<NUM_THREADS;i++)
            {
                sum+=tempPieces[i].size();
                state[i].clear();
            }
            share_mem.clear();
            #ifdef DEBUG
                cout << "After " << t << "s " << sum<< endl;
            #endif
        }
        return 0;
}
```

5.2.2 findShare

```
void *runPiece(void *threadarg)
{
    int tid = *((int*)threadarg);
    for(auto it=tempPieces[tid].begin();it!=tempPieces[tid].end();)
    {
        auto temp=it;
        int start=getPostion(it->x,it->y);
        run(*it);
        int end=getPostion(it->x,it->y);
        if(start!=end)
        {
            pthread_mutex_lock (&mutex);
            share_mem.push_back(*it);
            pthread_mutex_unlock(&mutex);
            list<Piece>::iterator temp=it++;
            tempPieces[tid].erase(temp);
        }
        else
        {
            it++;
        }
    }
}
```

5.2.3 dealPieces

```
void *doSplit(void *threadarg)
{

    int tid = *((int*)threadarg);
    for(auto it=tempPieces[tid].begin();it!=tempPieces[tid].end();++it)
    {
        int postion=getPos(it->x,it->y);
        if(state[tid].find(postion)==state[tid].end())
        {
            vector<list<Piece>::iterator> tmpvec{it};
            state[tid].insert(make_pair(postion,tmpvec));
        }
        else
        {
            state[tid][postion].push_back(it);
        }
    }
    unordered_map<int,vector<list<Piece>::iterator>>::iterator
item=state[tid].begin();
    for(item;item!=state[tid].end();item++)
     {
        if(item->second.size()>=2)
        {
            auto it=item->second.begin();
            auto mins=(*it)->speed;
            auto mind=(*it)->dirc;
            bool alive=true;
            ++it;
            for(;it!=item->second.end();++it){
                if((*it)->speed<mins){
                    mins=(*it)->speed;
                    mind=(*it)->dirc;
                    alive=true;
                }
                else if((*it)->speed==mins){
                    alive=false;
                }
                else{}
            }
            it=item->second.begin();
            if(alive){
```

```
                (*it)->speed=mins;
                (*it)->dirc=mind;
                ++it;
            }
            for(;it!=item->second.end();++it){
                int postion=item->first;
                tempPieces[tid].erase(*it);
            }
        }
    }
}
```

六、node8 结点单机运行结果

6.1 实验环境

内存配置如下：

```
[AU032@node8 ~/data]$ free m
             total       used       free     shared    buffers     cached
Mem:      16330940    1011376   15319564        356      53836     157404
-/+ buffers/cache:     800136   15530804
Swap:      4194300      46708    4147592
```

硬盘配置如下：

```
[AU032@node8 ~/data]$ df
Filesystem            1K-blocks      Used   Available Use% Mounted on
/dev/sda2              41153856   1983708    37072996   6% /
tmpfs                   8165468         0     8165468   0% /dev/shm
/dev/sda1                499656     43760      429684  10% /boot
inodell@o2ib0:/public
                     11483687136 326144932 10562006604   3% /public
```

CPU 情况如下：

共有 12 个处理器。

6.2 实验方法

为了节约实验的时间，采取的方式是，xshell 同时多个窗口连接。近乎同时开始执行单线程，双线程，四线程以及更多线程代码。由于同时试验多份代码，不可避免会导致效率降低。而且此种情况能尽量保证试验环境的一致性。

6.3 实验结果

线程数量	real	user	sys
1	12m44.156s	12m25.153s	0m3.084s
2	9m32.315s	13m19.341s	0m5.272s
4	7m30.596s	13m13.796s	0m8.084s
8	7m2.569s	12m37.244s	0m15.033s
12	8m2.925	14m35.240s	0m23.751s
串行	9m53.489s	9m50.896s	0m0.425s

Table1 多线程程序执行时间结果表(本机版本)

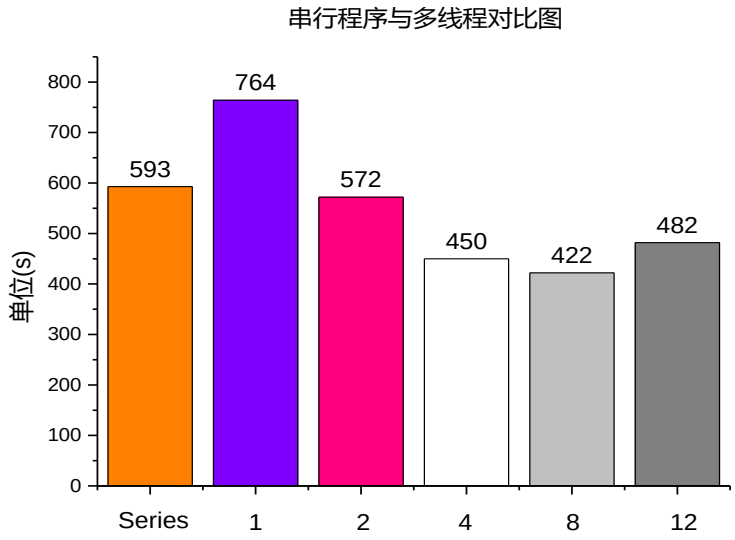


Fig 1. 串行程序与多线程程序执行时间结果图

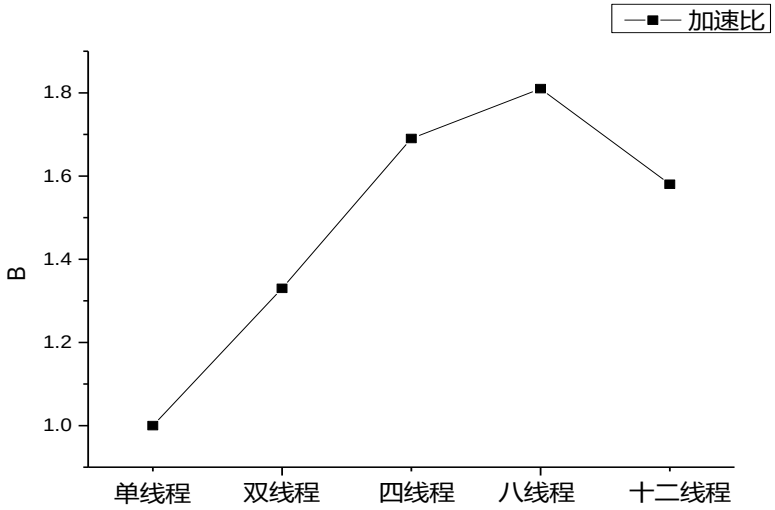


Fig 2. 多线程加速比图

从 Fig 1 中我们可以得出，当线程数为 2,4,8,12 时，多线程执行的效率都要高于串行程序。但是随着线程数量的增加，多线程程序执行减少的时间并不是固定的或者是越来越大的。而是越来越小。甚至超出了一定的范围后，多线程执行的时间反而会增加。

七、qsub 实验结果

首先写好了各个线程的 pbs 文件，然后同时执行这些 pbs 文件，并将结果输出到对应的 run.log 文件中去。ppn 的数量和线程的数量保持一致。

```
node5:
                                                                  Req'd     Req'd       Elap
Job ID                  Username    Queue    NDS   TSK    Memory    Time      S Time      BIG  FAST   PFS
----   ---------------- ----------- -------- ----- ------ --------- --------- - --------- ----- ----- -----
154146.node5            AU032       qstudent     1      1       --   00:20:00 R  00:05:27   --    --    --
154147.node5            AU032       qstudent     1      2       --   00:20:00 R  00:05:24   --    --    --
154148.node5            AU032       qstudent     1      4       --   00:20:00 R  00:05:08   --    --    --
154149.node5            AU032       qstudent     1      8       --   00:20:00 R  00:05:08   --    --    --
154150.node5            AU032       AU032_qu     1     12       --        -- Q       --    --    --    --
```

运行结束后，文件结构如下所示：

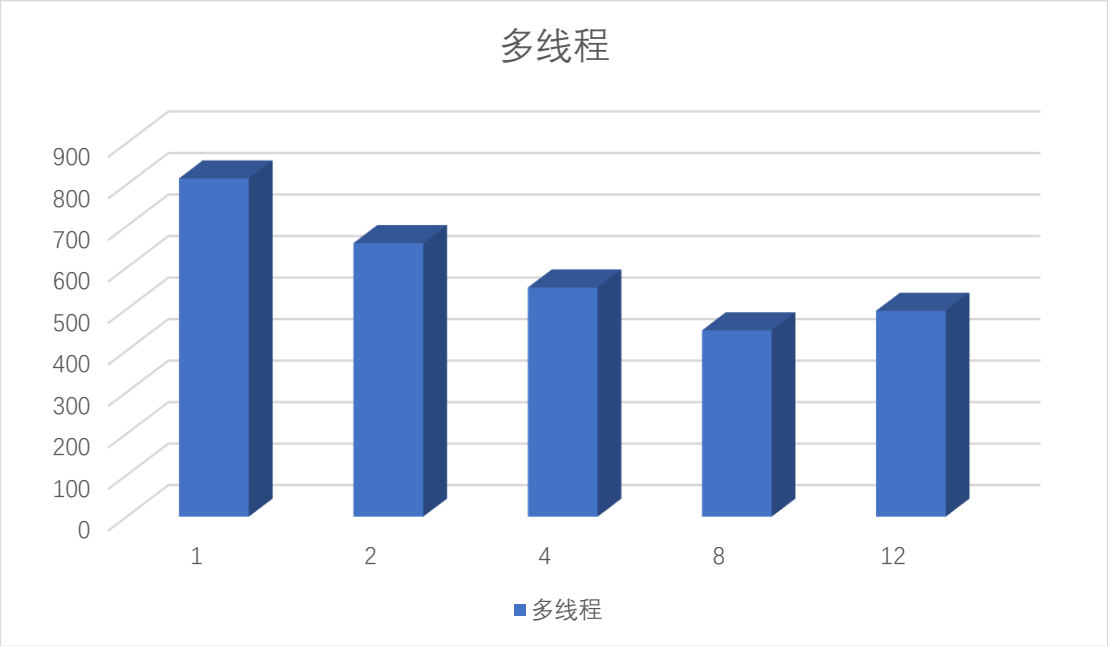
```
[AU032@node8 ~/data/finalPthread]$ ls -l
total 9412
-rw-r--r-- 1 AU032 users    6180 Nov 12 12:55 finalPthread12.cpp
-rw-r--r-- 1 AU032 users    6179 Nov 12 12:53 finalPthread1.cpp
-rw-r--r-- 1 AU032 users    6179 Nov 12 12:54 finalPthread2.cpp
-rw-r--r-- 1 AU032 users    6179 Nov 12 12:55 finalPthread4.cpp
-rw-r--r-- 1 AU032 users    6179 Nov 12 12:55 finalPthread8.cpp
-rw-r--r-- 1 AU032 users    6179 Nov 12 12:52 finalPthread.cpp
-rw-r--r-- 1 AU032 users 7246257 Nov  7 19:59 input.txt
-rwxr-xr-x 1 AU032 users   82395 Nov 11 19:28 main
-rwxr-xr-x 1 AU032 users   84988 Nov 12 12:55 main1
-rwxr-xr-x 1 AU032 users   82397 Nov 12 12:56 main12
-rw------- 1 AU032 users      46 Nov 15 21:05 main12.o154150
-rw------- 1 AU032 users      46 Nov 15 20:52 main1.o154146
-rwxr-xr-x 1 AU032 users   84988 Nov 12 12:56 main2
-rw------- 1 AU032 users      46 Nov 15 20:54 main2.o154147
-rwxr-xr-x 1 AU032 users   84988 Nov 12 12:56 main4
-rw------- 1 AU032 users      46 Nov 15 20:50 main4.o154148
-rwxr-xr-x 1 AU032 users   82396 Nov 12 12:56 main8
-rw------- 1 AU032 users      45 Nov 15 20:48 main8.o154149
-rw-r--r-- 1 AU032 users     150 Nov 15 20:41 p12.pbs
-rw-r--r-- 1 AU032 users     146 Nov 15 20:40 p1.pbs
-rw-r--r-- 1 AU032 users     146 Nov 15 20:40 p2.pbs
-rw-r--r-- 1 AU032 users     146 Nov 15 20:40 p4.pbs
-rw-r--r-- 1 AU032 users     146 Nov 15 20:41 p8.pbs
-rw-r--r-- 1 AU032 users  349502 Nov 15 21:05 run12.log
-rw-r--r-- 1 AU032 users  349502 Nov 15 20:52 run1.log
-rw-r--r-- 1 AU032 users  349502 Nov 15 20:54 run2.log
-rw-r--r-- 1 AU032 users  349502 Nov 15 20:50 run4.log
-rw-r--r-- 1 AU032 users  349502 Nov 15 20:48 run8.log
```

各个文件执行结果如下表所示：

线程数量	花费时间
1	real 13m35.370s user 18m34.749s sys 0m9.151s
2	real 11m18.794s user 11m11.914s sys 0m5.153s
4	real 9m12.325s user 20m23.997s sys 0m14.957s
8	real 7m29.837s user 22m0.889s sys 0m30.135s
12	real 8m16.235s user 25m37.021s sys 0m58.952s

Table2 多线程程序执行时间结果表(qsub 版本)

为直观的得到表结果，将表格内容作图如下：



由上图可以看到，随着线程数量的增加，执行程序的时间(s)逐渐变小，但是越过了 8 个线程之后执行时间反而会反弹。再次验

证了本地单机的试验结果。

八、理论实验结果与性能分析

8.1 理论实验结果

首先计算在总迭代次数中各个主要部分花费的时间。

串行程序执行时间 T_S 为:

T_S 的主要构成部分为三块，读取文件保存到 list<piece>花费的时间 T1；遍历 list 并把数据添加到 state 的时间 T2；以及在 state 中将数据进行处理花费的时间 T3。

在实验中，串行程序执行的 T1,T2,T3 分别为：0.47s, 511s, 82s。

理论结果计算公式如下:

$$T = \frac{单线程执行时间}{线程数量} \quad (1)$$

Table3．理论结果与实际结果对比表

线程数量	理论结果	实际结果
1	764	764
2	382	572
4	191	450
8	95.5	422
12	64	482

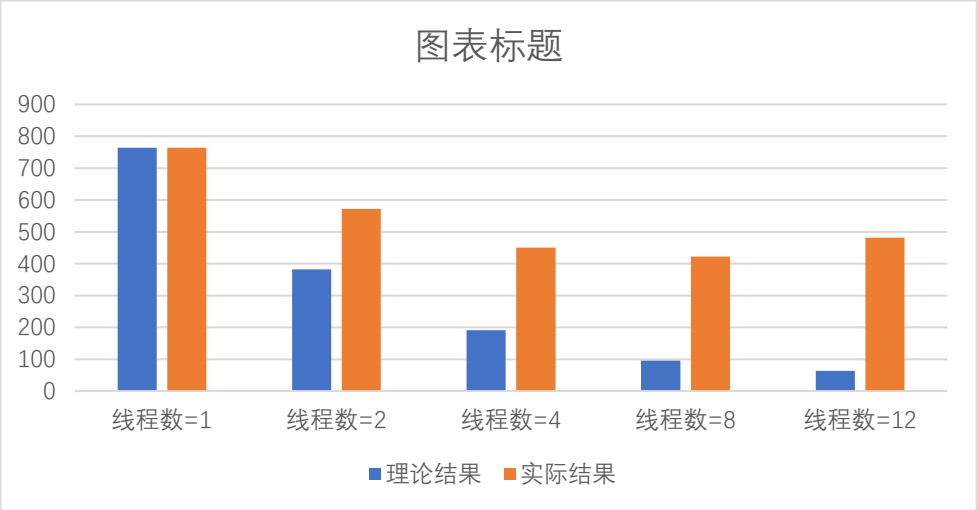


Fig 3. 理论结果与实际结果对比图

假设单线程的程序执行时间为 t 秒，那么理论上最好的双线程程序执行时间应该为约等于$\frac{t}{2}$，最好的四个线程执行程序的时间应该为$\frac{t}{4}$，并以此类推。但显然实际实践结果表明并不符合这个要求。

首先，我们很难做的到将整个串行程序改造为完全的并行程序，我们只能适当的将其中的一部分进行并行。

其次，线程之间互相调用也需要花费时间，不同时刻，cpu 状态不相同，处理时间也会有差异。

最后，并行程序划分的合理性，共享锁的设置，硬件环境的配置都会对多线程结果造成影响。因此理论上的效果很难达到。

8.2 性能分析

针对本数据集以及本实验环境下来看，显然采用 8 个线程，程序执行效果最好。如果在 8 的基础上继续增加线程。那么效率不仅不会提高。反而会降低。因为任务过多，就会导致一个 cpu 处理的线程大于一个。从而降低执行效率。但是随着线程数量的增加 cpu 的利用率是

逐渐增加的。因此我们认为即使是在单 cpu 状态下使用多线程技术也是非常有作用的。能充分利用 cpu 资源。并给用户带来更好的体验效果。由于 cpu 处理器的限制，程序执行的时间不能一直随着线程数量的增加而减小。因为会出现同一个 cpu 轮转执行线程的状况发生。这就导致执行时间变大。因此一定要根据硬件环境合理的划分线程数量。

八、总结与展望

在多线程解决实际问题的过程中，大大提高了自己的代码能力，以及将串行代码改造成为并行代码的能力。在该实验中不断的测试了各种数据分割方法以及功能分割方法，对 pthread 的各种使用方法更加了解。知道如何对串行任务进行划分，并进行分析。

实验最终选择的了根据区域划分的方法，很好的实现了提升性能。

希望在以后的生活中能合理根据硬件环境以及任务描述，将任务做到更加合理的分解。

附录：

Pthread 编译指令：

```
g++ -std=c++0x -o main *.cpp -lpthread
```

程序执行指令：

```
time ./main
```

输出代码段执行时间：

```
timeval tBegin, tEnd, Diff_time;
gettimeofday(&tBegin, NULL);
long start = ((long)tBegin.tv_sec)*1000+(long)tBegin.tv_usec/1000;
gettimeofday(&tEnd, NULL);
long end = ((long)tEnd.tv_sec)*1000+(long)tEnd.tv_usec/1000;
cout<<"时间： "<<(end - start)<<" ms"<<endl;
```