[](http://www.tju.edu.cn/)

并行计算实验报告

[](https://image.baidu.com/search/detail?ct=503316480&z=undefined&tn=baiduimagedetail&ipn=d&word=%E5%A4%A9%E6%B4%A5%E5%A4%A7%E5%AD%A6%E6%A0%A1%E5%BE%BD&step_word=&ie=utf-8&in=&cl=2&lm=-1&st=undefined&cs=276026180,1083048773&os=1375696740,2873564260&simid=3431510383,517316568&pn=1&rn=1&di=178729471910&ln=271&fr=&fmq=1541992853406_R&fm=&ic=undefined&s=undefined&se=&sme=&tab=0&width=undefined&height=undefined&face=undefined&is=0,0&istype=0&ist=&jit=&bdtype=0&spn=0&pi=0&gsm=0&hs=2&objurl=http://www.tju.edu.cn/zsn/yxjs/yxsz/201606/W020160620590466512014.png&rpstart=0&rpnum=0&adpicid=0)

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 3](#_Toc530083329)

[二、 实验内容与多线程介绍 3](#_Toc530083330)

[2.1 实验内容 3](#_Toc530083331)

[2.2多线程介绍 4](#_Toc530083332)

[2.2.1多线程出现的原因 4](#_Toc530083333)

[2.2.2 多线程的优缺点 4](#_Toc530083334)

[2.2.3多线程提高执行效率 5](#_Toc530083335)

[三、研究过程 5](#_Toc530083336)

[四、程序流程图 7](#_Toc530083337)

[五、伪代码以及具体实现 7](#_Toc530083338)

[5.1伪代码 7](#_Toc530083339)

[5.2关键代码 8](#_Toc530083340)

[5.2.1 main函数 8](#_Toc530083341)

[5.2.2 findShare 10](#_Toc530083342)

[5.2.3 dealPieces 11](#_Toc530083343)

[六、node8结点单机运行结果 12](#_Toc530083344)

[6.1 实验环境 12](#_Toc530083345)

[6.2 实验方法 13](#_Toc530083346)

[6.3实验结果 13](#_Toc530083347)

[七、qsub实验结果 15](#_Toc530083348)

[八、理论实验结果与性能分析 17](#_Toc530083349)

[8.1理论实验结果 17](#_Toc530083350)

[8.2性能分析 18](#_Toc530083351)

[八、总结与展望 19](#_Toc530083352)

[附录： 20](#_Toc530083353)

# 题目描述

使用pthread实现荒野求生，并进行性能分析。

# 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。

# 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。

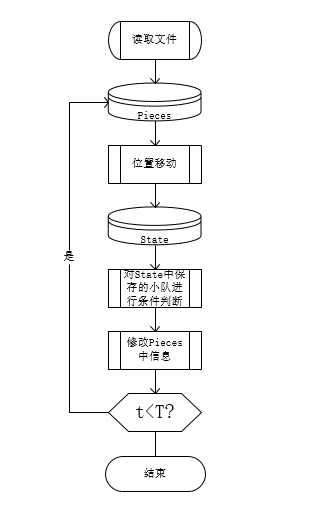
发现实际的问题在于每一个子list执行的时间都大于串行代码中整个list的执行时间，经过测试发现是互斥锁存在问题。导致多线程绝大多数部分不能并行执行，只能串行执行。因此会表现出随着线程增多，并行程序执行时间变长的特征。

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。

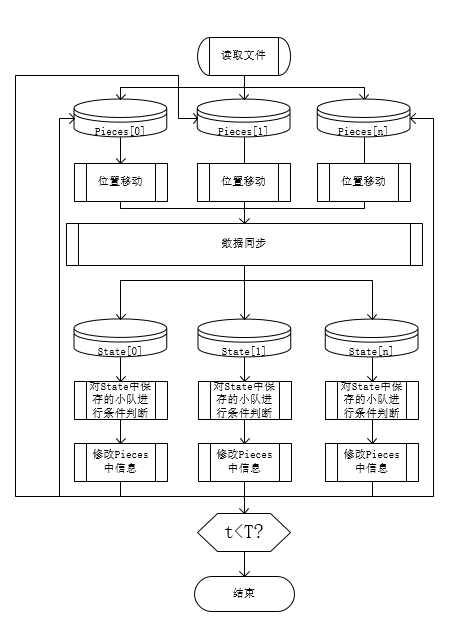
* 最后想到的也是最终采用的方法是按照小队所属区域进行划分。按照一个小队的x坐标和y坐标，并将x,y两个坐标映射到一维向量中，对所有小组进行线程分配，经过实际的测试验证发现，该方法的效果更加好。

# 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);

}

}

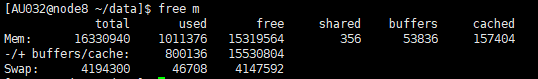
}

}

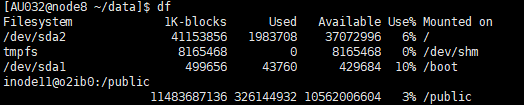
# 六、node8结点单机运行结果

## 6.1 实验环境

内存配置如下：



硬盘配置如下：



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

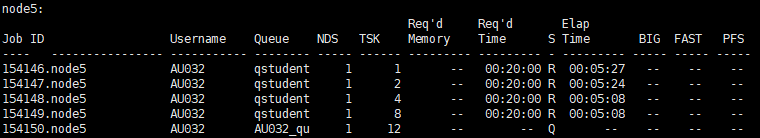


Fig 2. 多线程加速比图

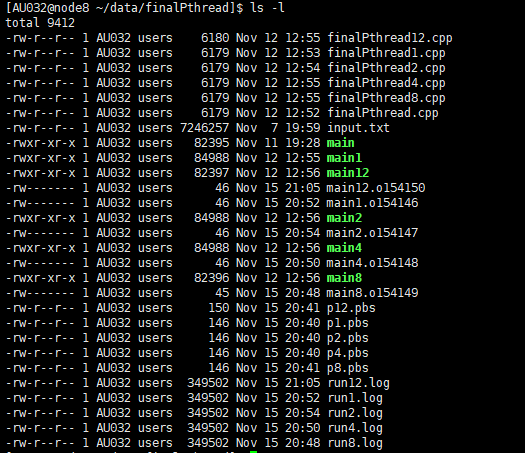
从Fig 1中我们可以得出，当线程数为2,4,8,12时，多线程执行的效率都要高于串行程序。但是随着线程数量的增加，多线程程序执行减少的时间并不是固定的或者是越来越大的。而是越来越小。甚至超出了一定的范围后，多线程执行的时间反而会增加。

# 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。



运行结束后，文件结构如下所示：



各个文件执行结果如下表所示：

|  |  |
| --- | --- |
| 线程数量 | 花费时间 |
| 1 |  |
| 2 |  |
| 4 |  |
| 8 |  |
| 12 |  |

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TS为:

TS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：

（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，最好的四个线程执行程序的时间应该为，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;