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 MPI 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 MPI 技术实现荒野求生，并进行性能分析。

MPI 是一个跨语言的通讯协议，用于编写并行计算机。支持点对点和广播。MPI 是一个信息传递应用程序接口，包括协议和和语义说明，他们指明其如何在各种实现中发挥其特性。MPI 的目标是高性能，大规模性，和可移植性。MPI 在今天仍为高性能计算的主要模型。

Mpi 为一次代码多次执行的并行计算模型，是多进程的实现方式。

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 MPI 介绍

MPI 并不是一门语言。MPI 只是一个标准，这个标准定义了核心库的语法和语义，这个库可以被 Fortran 和 C 调用构成可移植的信息传递程序。MPI 提供了适应各种并行硬件商的基础集，他们都被有效的实现。这导致了是硬件商可以基于这一系列底层标准来创建高层次的惯例，从而为分布式内存交互系统提供他们的并行机。MPI 提供了一个简单易用的可移植接口，足够强大到程序员可以用它在高级机器上进行进行高性能信息传递操作

2.2.1 多进程 MPI 与多线程 Pthread 的区别

进程和线程的区别归纳：

1. 地址空间和其它资源：进程间相互独立，同一进程的各线程间共享。某进程内的线程在其它进程不可见；
2. 通信：进程间通信 IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信（需要进程同步和互斥手段的辅助，以保证数据的一致性）；
3. 调度和切换：线程上下文切换比进程上下文切换要快得多。

2.2.2 MPI 传输的数据类型

初始做代码设计的时候，希望直接用 MPI 传输对象，但实际发现 MPI 并不支持直接传输对象，MPI 中直接支持的数据类型有 float, int, double, char,以及数组。

也可以传输结构体。但是传输方式较为复杂，经过调研发现可能可以使用 boost 将类或者数据序列化后再进行数据传输。但是过程比较复杂，因此最后采用了数组传输，再将数组构造回对象使用的方式来进行数据传输。

2.2.3MPI 的编程模式与通信方式

MPI 并行编程模式主要有两种：

- 对等模式—程序的各个部分地位相同,功能和代码基本一致,只是处理的数据或对象不同。
- 主从模式—程序通信进程之间的一种主从或依赖关系

MPI 通信模式主要也只有两种：

- 点对点通信模式
- 组通信模式

本次实验中采用了，主从编程模式以及点对点通信方法。

2.2.4 MPI 常用的函数

MPI_Init(…)

MPI_Comm_size(…)

MPI_Comm_rank(…)

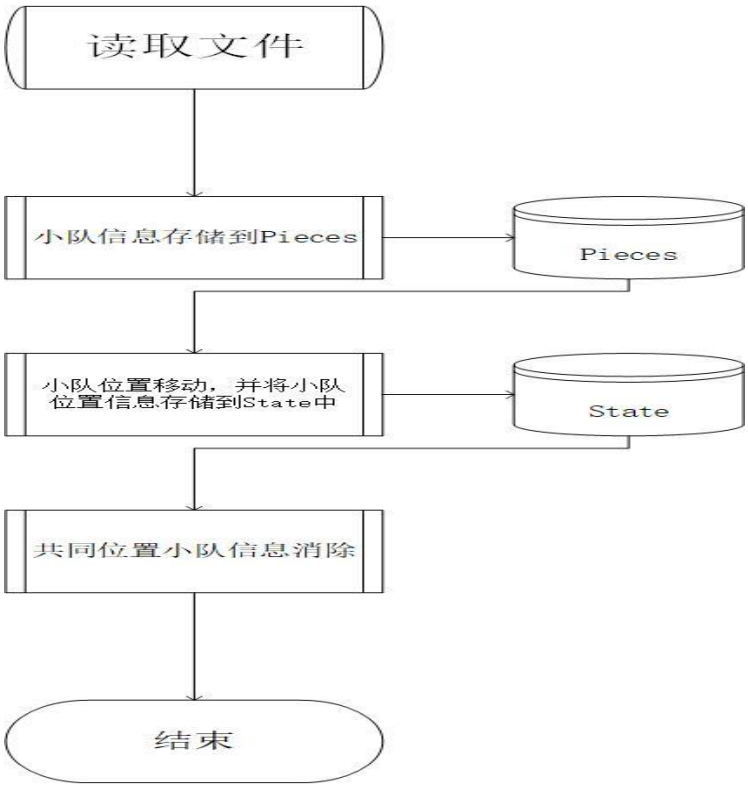
MPI_Send(…)

MPI_Recv(…)

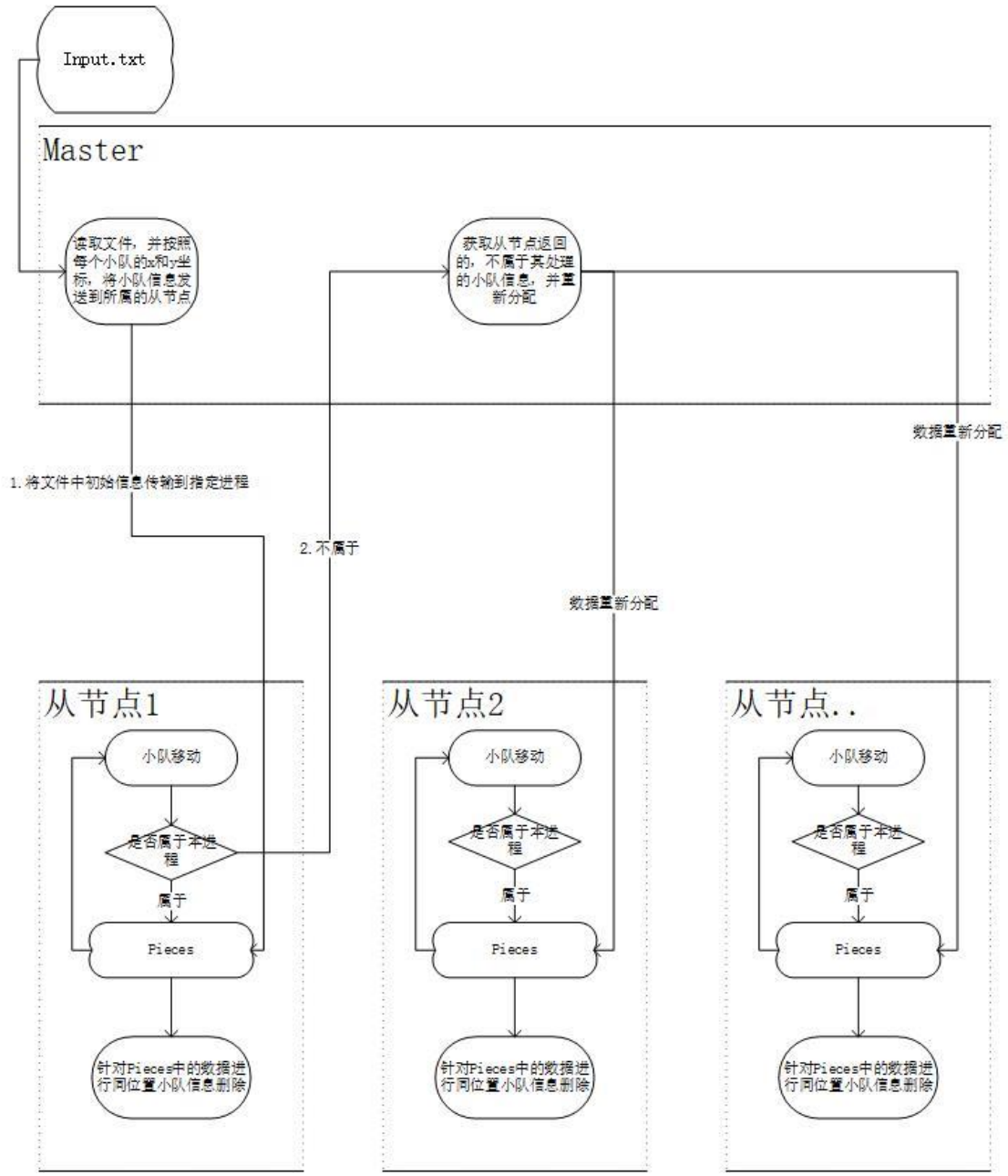
MPI_Finalize()

三、程序流程图

3.1 串行程序执行流程图



3.2 MPI 程序执行流程图



图中绘制的是一次执行过程，第一次将小队信息存储到从节点的 piece 中后，进行小队位置移动，移动了的小队如果不再属于本进程处理的区域，那么就把该节点传输回 master 节点进行重新分配。以后的多次迭代，也就是不断的判断每个从节点中的小队是否移动出了本进程处理的范围，不停的迭代执行。

四、研究过程

从一开始就确定了实验的思路。首先 master 节点读取 input 文件之后按照每个小队所处的位置。划分为从节点数量个区域，之后将每个区域的数据发送给对应的从节点，从节点首先将获得的数据进行移动。移动一次以后，有的节点可能不再属于本节点能处理的区域范畴，那么需要把这个小队信息发送给 master 节点进行重新分配，每个从节点将数据发送给 master 节点后，master 将收到的这些数据重新分配给每个从节点。每个从节点做小队的同位置消除工作。

确定了思路以后，仍有很大的问题存在。由于 MPI 不支持传输对象，所以思路不需要更换。但是，我尝试了**三种实现方**法，这里会一一列举。

4.1 完全重构代码（程序只使用数组）

产生这种思路最主要的原因在于考虑不周到，单纯的使用数组来做数据传输是比较好的思路。但是如果传输到从节点之后只使用数组来进行处理，有几个问题很麻烦。

- ✧ 以数组的方式传输到从节点之后，确定了哪些小队需要被删除之后，对数组进行删除非常的麻烦。因为数组删除，只能将要索引位置之后的数据全部向前提一个数据单位。但是这样花费的时间代价过高。由于删除次数比较大。
- ✧ 针对删除代价高的问题想到，可以维护一个删除信息数组，这样每次不用真的在从节点中将数组中的某些小队真正的删除，

只需要将这些小队加入到删除信息数组中就可以。但是这样又面临一个问题，由于不能真正的删除数组中的信息，导致 MPI 中每次传输的数据量不会减小，又导致 MPI 信息传输时间过高。

4.2 完全重构代码(不单纯使用数组)

在 3.1 中提出的两个问题想到使用动态数组 vector 来进行数据删除。发送过程使用固定长度数组进行传输，但是数据传输到从节点之后，将固定长度数组保存成 Vector 形式。这样解决掉了数据删除的问题。

但是这样又存在一个致命的问题。由于使用数组传输，数组中每个小队的索引相当于就是在小队中所处的位置，例如 x[1]代表第一小队的 x 轴坐标。这个 1，代表这个小队的索引，也代表了这个小队在数组中的位置信息，如果我们想把小队信息在数组中进行删除，那么如果删除了原来的第 a 个小队，那么 a 之后的所有小队，位置信息都会发生改变。假设第一次需要删除标号为 a 的小队，第二次需要删除标号为 b 的小队，并且 b>a,如果按照先删除 a 再删除 b 小队的方法来做会出错，因为删除了 a 之后，b 小队的实际信息不再存储于 b 位置。而是 b-1 这个位置。如果想维护这样的位置信息，代价过于高昂。因此这样方法也被放弃。

4.3 半重构代码

数据传输使用数组，但是数据传输到从节点之后，将数组信息在构造回 Piece 类。这样能保证最大程度的串行代码使用

五、伪代码

5.1 伪代码

```
begin
    data <- readFile(intput.txt)
    for each team in data
    do
        MPI_Send(team,指定从节点)
    end for

    if rank=master then
        for t in 0 to T
        do
            Recv(reAllocTeam,所有从节点)
            and
            MPI_Send(team,所属从节点)
            Recv(从节点剩余小队数量信息)
        end for
    endIf
    elseIf rank=从节点 then
        Recv(小队信息)
        run(小队)
        if 小队不属于本进程 then
            send(小队，master)
        endIf
        Recv(master 返回的共享变量中的小队信息)
        共同位置小队信息删除
    endIf

end
```

六、Node8 节点本机测试结果

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
inode11@o2ib0:/public
                     11483687136 326144932 10562006604   3% /public
```

CPU 情况如下：

共有 12 个处理器。

```
[AU032@node8 ~/data]$ cat /proc/cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 16
model           : 8
model name      : Six-Core AMD Opteron(tm) Processor 2435
stepping        : 0
cpu MHz         : 2600.000
cache size      : 512 KB
physical id     : 0
siblings        : 6
core id         : 0
cpu cores       : 6
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht sysca
lm 3dnowext 3dnow constant_tsc rep_good nonstop_tsc extd_apicid pni monitor cx16 popcnt lahf_lm cmp_legacy svm extapic cr8_l
etch osvw ibs skinit wdt npt lbrv svm_lock nrip_save pausefilter
bogomips        : 5226.91
TLB size        : 1024 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management: ts ttp tm stc 100mhzsteps hwpstate
```

6.2 实验方法

此次试验并未采取 xshell 中同时开多个窗口执行程序的方法，而是单个执行，查看效果。

由于本实验采用的是主从结点方式来进行程序处理，因此实际如果设置了 mpirun -np 8 但是其中真正执行数据删除处理的结点其实只有 7。

6.3 实验结果

进程数量	time
2	7m32s
4	2m43s
6	1m32s
8	1m8s
10	57s
12	1m57s
14	1m37s
串行	7m6s

Table1 多线程程序执行时间结果表

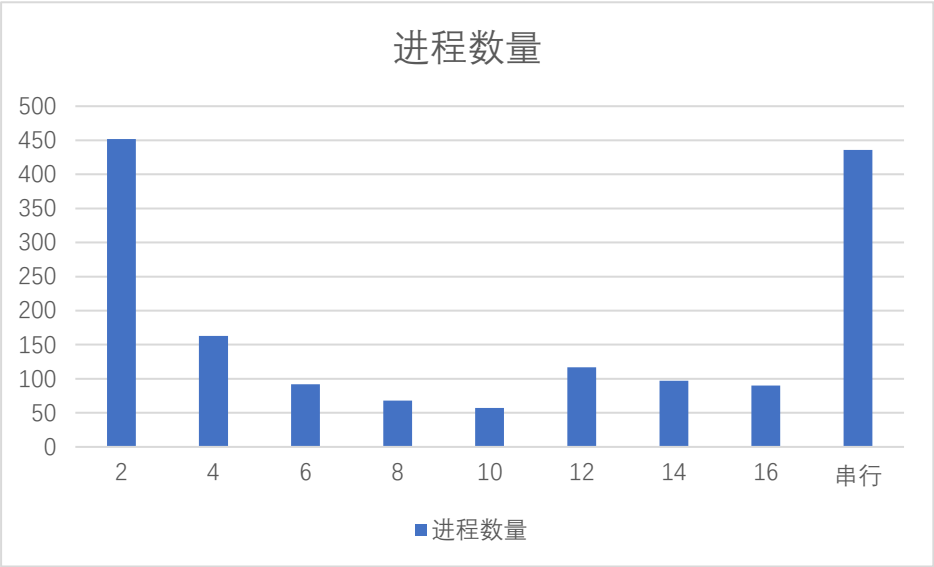


Fig 1. 串行程序与多进程程序执行时间结果图

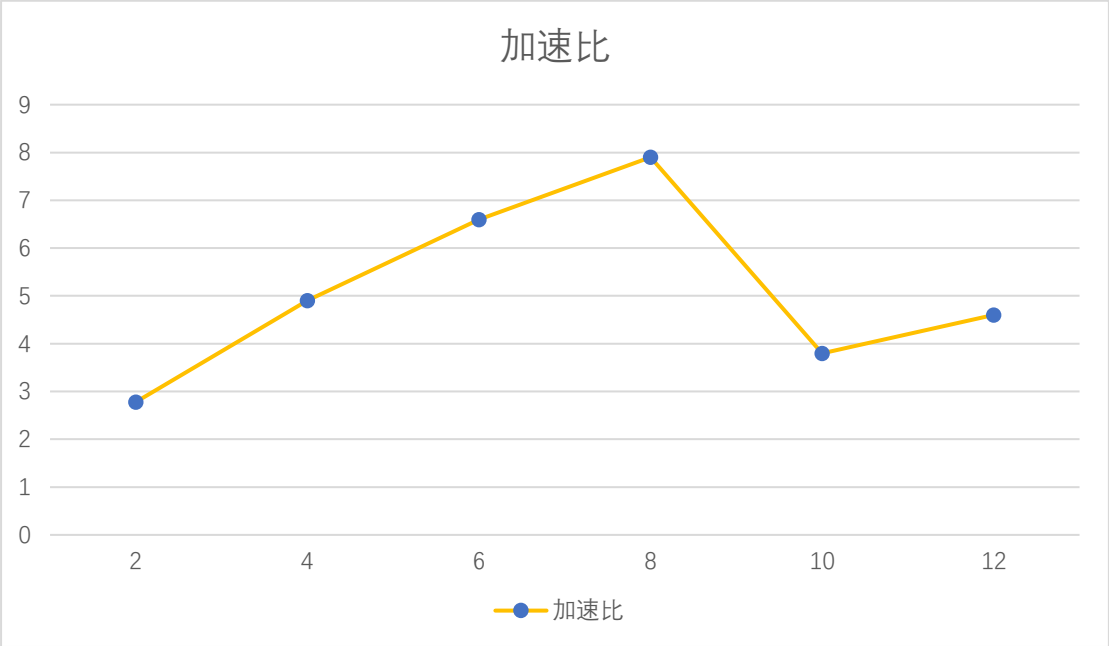


Fig 2. 多进程加速比图

从 Fig 1 中我们可以看出，采用多进程的时候，只有采用两个进程的时候，执行时间大于串行程序执行时间。因为两个进程的时候，实际只有一个进程在进行数据处理，相当于在串行程序执行的基础上额外增加了许多信息传递的消耗。

当进程数为，4/6/8/10/12/14 时候，多进程执行的时间都要小于串

行程序执行时间。但是在进程数为 10，取得了多进程的最优数值。越过了这个最优数值之后，虽然数据有一些小的波动，但是效果都比这一个差。因此我们可以看到，进程数并不是越多越好的。而是有一个合适的阈值。

从 Fig 2 中我们可以明显的看出，加速比实际上是在下降的。但是加速比下降没有多线程那么明显。

七、qsub 实验结果与分析

首先根据具体内容写好pbs文件,之后提交。提交结果界面如下所示：

```
                                                  Req'd    Req'd       Elap
Job ID                  Username    Queue    NDS   TSK    Memory    Time      S Time       BIG  FAST   PFS
----  ----------------- ----------- -------- ----- ------ --------- --------- - ---------- ----- ----- -----
154192.node5            AU032       qstudent     1      6       --   00:20:00 C       --     --    --    --
154193.node5            AU032       qstudent     1      8       --   00:20:00 C       --     --    --    --
154194.node5            AU032       qstudent     1     10       --   00:20:00 C       --     --    --    --
154195.node5            AU032       qstudent     1     12       --   00:20:00 C       --     --    --    --
154196.node5            AU032       qstudent     1     14       --   00:20:00 C       --     --    --    --
```

提交所有程序执行完毕之后会返回对应的时间结果文件以及 run.log 文件，生成完结果之后的文件结构如下所示：

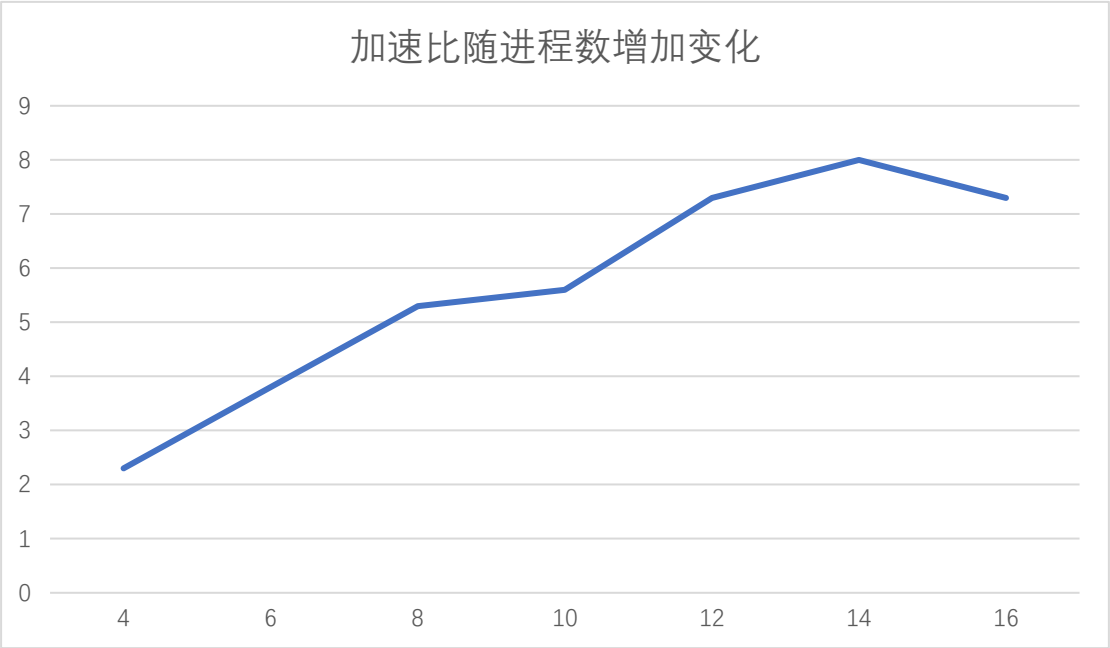
```
[AU032@node8 ~/data/mpi]$ ls -l
total 9636
-rw-r--r-- 1 AU032 users   10403 Nov 13 22:56 4MPI.cpp
-rw-r--r-- 1 AU032 users    9784 Nov 14 08:56 4MPItestArea.cpp
-rw-r--r-- 1 AU032 users   14243 Nov 14 10:43 finalMPI.cpp
-rw-r--r-- 1 AU032 users    7673 Nov 14 18:28 finalVersion.cpp
-rw-r--r-- 1 AU032 users 7246257 Nov 13 11:56 input.txt
-rwxr-xr-x 1 AU032 users  184530 Nov 14 18:28 main
-rw------- 1 AU032 users    1513 Nov 15 21:21 main10.o154194
-rw------- 1 AU032 users    1771 Nov 15 21:19 main12.o154195
-rw------- 1 AU032 users    2027 Nov 15 21:20 main14.o154196
-rw------- 1 AU032 users     744 Nov 15 21:16 main4.o154184
-rw------- 1 AU032 users    1002 Nov 15 21:19 main6.o154192
-rw------- 1 AU032 users    1258 Nov 15 21:22 main8.o154193
-rw-r--r-- 1 AU032 users     184 Nov 15 21:18 mp10.pbs
-rw-r--r-- 1 AU032 users     184 Nov 15 21:18 mp12.pbs
-rw-r--r-- 1 AU032 users     184 Nov 15 21:18 mp14.pbs
-rw-r--r-- 1 AU032 users     180 Nov 15 21:16 mp4.pbs
-rw-r--r-- 1 AU032 users     180 Nov 15 21:18 mp6.pbs
-rw-r--r-- 1 AU032 users     180 Nov 15 21:18 mp8.pbs
-rw-r--r-- 1 AU032 users    6858 Nov 14 01:57 MPI.cpp
-rw-r--r-- 1 AU032 users    9379 Nov 13 13:22 mpi_transData.cpp
-rw-r--r-- 1 AU032 users    2403 Nov 12 21:59 mpi_transDataTemp.cpp
-rw-r--r-- 1 AU032 users  349991 Nov 15 21:21 runMi10.log
-rw-r--r-- 1 AU032 users  350006 Nov 15 21:19 runMi12.log
-rw-r--r-- 1 AU032 users  350022 Nov 15 21:20 runMi14.log
-rw-r--r-- 1 AU032 users  349944 Nov 15 21:16 runMi4.log
-rw-r--r-- 1 AU032 users  349960 Nov 15 21:19 runMi6.log
-rw-r--r-- 1 AU032 users  349975 Nov 15 21:22 runMi8.log
-rwxr-xr-x 1 AU032 users  184513 Nov 14 11:11 test
```

打开对应的执行文件查看程序执行时间，最终得到表格如下：

进程数	执行时间
4	real 3m3.084s user 12m7.054s sys 0m2.499s
6	real 1m50.753s user 10m58.722s sys 0m2.875s
8	real 1m19.292s user 10m26.011s sys 0m2.475s
10	real 1m16.342s user 12m9.014s sys 0m3.365s
12	real 0m58.324s user 11m27.712s sys 0m3.473s
14	real 0m53.915s user 12m21.188s sys 0m3.312s
16	real 0m58.138s user 15m11.497s sys 0m6.289s

使用 qsub 提交，并指定 ppn 与进程数保持一致时。程序的执行效率一直在提高。但是提高的程度越来越低。到了 16 个进程时加速比反而开始下降。因此问题与多线程一致。实际进程的设计数量应该考虑本地硬件环境来进行设计。

加速比曲线如图所示：



从加速比曲线可以看出，16 进程以前，加速比随着进程数量的增加而增加。但是到了 16 个进程后发现程序的执行效果反而下降。而且很明显可以看出，多线程加速比的整体提高效率是逐渐降低的。

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。

Table2. 理论结果与实际结果对比表

进程数量	理论结果	实际结果
4	106	163
6	71	92
8	53	68
10	42.6	57
12	35.5	117
14	30	97

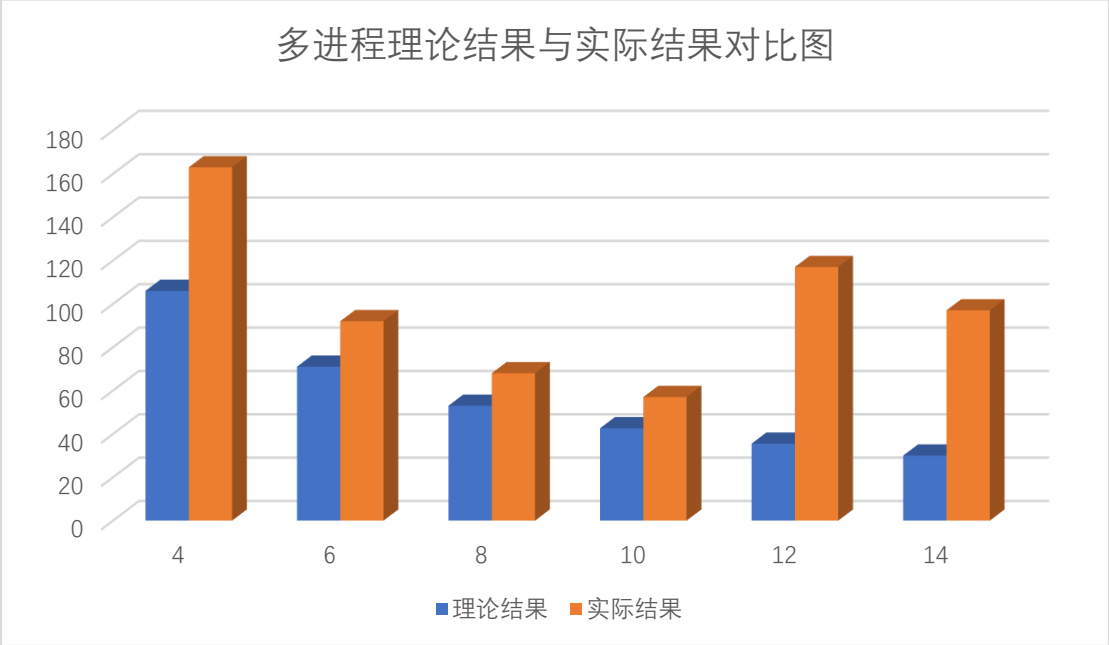


Fig 3. 理论结果与实际结果对比图

从 Fig 3 中我们可以看出，与多线程相比，多进程对串行程序的优化程度更加高，也就是说多进程程序实际执行时间距离理论执行时间更为接近。

8.2 性能分析

从 Table 2 和 Fig3 中我们可以看出，当进程数量为 10 的时候，取得了在 node8 结点环境下的最优值，但是使用 qsub 时在 14 进程数取得了当前数据集下多进程的的最优值。但是两种环境下都表现出了数据变化的一致性，当进程数量超过某个界限的时候，多进程执行时间依旧小于串行执行时间，但是终究会低于串行时间。因此一定要根据现有环境选定合适的进程数量。

九、MPI 与多线程对比

在本次实验中发现，在当前试验环境以及数据集合下，多进程的表现效果大大超过多线程。经过分析后发现是我的划分方法更加适用于多进程来做。

因为 MPI 中每一个进程都拥有自己的一份数据，不存在锁这样的东西。但是多线程中采用全局变量来作为共享空间难免会需要加锁解锁。而这种设置了锁的位置就会导致并行程序排队串行执行。

十、总结与展望

在本次 MPI 实现并行程序的过程中，对 MPI 实现多进程的方法大大加深了了解。但是对于程序的实际执行仍旧有些疑问。因为实际执行的时候没有办法保证两次试验的进行是在一模一样的环境下，因此实际程序执行时间总是在变动的。只能多次试验取得平均数值。

遗憾的是本次试验没有做足够多数量的试验。希望在接下来的时间能做更多的试验，来弥补自己的学习情况。

附录：

MPI 编译指令：

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

MPI 程序执行指令：

```
Mpirun -np x ./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;
```