石邢越 16337208

张永东教授

高性能计算程序设计基础

2018 年 11 月 17 日星期六

实验 6: MPI 并行正则采样排序

1 实验目的

掌握正则采样排序算法，进一步加强 MPI 的并行程序设计能力。

2 实验要求

- 用 MPI 实现并行的正则采样排序。
- 待排序的数据为给定文件中的2^{31}个自然数。
- 要求完成并行正则采样排序的四个阶段中的前三个，即本地数据排序、获得划分主元(pivots)和交换数据(要求使用 MPI 集合通信)；不要求完成最后一步，即归并排序。
- 分别使用 1、2、4、8、16、32、64 和 112 核的资源完成排序，对比各种情况下的加速比和效率。

3 算法原理

3.1 并行正则采样排序算法

并行正则采样排序算法 (Parallel Sorting by Regular Sampling, PSRS) 是一种 1993 年由 Li etc. 提出的并行排序算法[1]。Li 的文章中，提到这一算法的优点包括：

- 使用与异构的 MIMD 体系结构，对硬件的要求较宽松；

- 有良好的负载平衡；
- 进程间的通信量不算特别大，可以接受；
- 具有较好的空间局部性。

3.2 MPI 与 C++实现的 PSRS 算法

PSRS 算法的过程主要可以分为以下 6 个阶段：

3.2.1 阶段 1——初始化

使用 MPI 前先进行常规的初始化，声明通信空间，为接下来的工作做准备。

在这一阶段，0 号进程还需要读取文件得到待排序的数据，读取文件的函数如下：

```
1  void reader(unsigned long* data, int size_out){
2        unsigned long element;
3        int counter = 0;
4        //cout << "size=" << sizeof(element) << endl;
5        ifstream fo("/share/home/shared_dir/psrs_data", ios::binary);
6        if (fo.is_open()) {
7              cout << "Open successfully" << endl;
8              while (!fo.eof()) {
9                    counter++;
10                   fo.read((char*)&element, 8);
11                   //cout  << element << endl;
12                   *(data+counter)=element;
13             }
14             size_out=counter;
15             return ;
16       }
17       else
18             cout << "Cannot open this file!" << endl;
19             return ;
20 }
```

代码 1 读取数据文件的函数

要注意数据文件为二进制文件，在声明 fstream fo 的时候要指明类型。

在不同的操作系统与机器上，unsigend long 的位数可能不同，在 Linux 系统上使用以上函数可以正常工作，但在 Windows 系统下声明成 unsigned __int64 会更可靠。

在与同学交流的过程中，我了解到了另外两种读取数据文件的方法，这两种算法都是让各个进程并行读取数据的。第一种是在 fstream 的 read 函数中

指明从文件的什么位置开始读以及需要读取的数据量；第二种利用了 MPI 中的并行读函数 MPI_File_read_ordered。这两种读取数据文件的方式比我采用的单进程读取使用了更少的数据通信（因为之后不再需要进程 0 分发数据），而且很可能可以减少读文件的时间（这取决于 read 函数和 MPI_File_read_ordered 的具体实现，我不知道它们是怎么将指针移动到特定位置的，所以这只是一个猜测）。

3.2.2 阶段 2 —— 本地数据排序和正则采样

这一阶段 0 号进程 scatter 在上一阶段读取的待排序数据（如果用之前讲到的并行读取这里就可以免去大量的通信代价）。

各个进程得到了本地数据后，使用 quicksort 进行本地数据排序，调用了 C++标准库中的 sort 函数。

之后在每个进程中采样出#procs 个样本，作为下一阶段划分主元的备选。

上述过程的代码如下：

```
1  //scatter unsorted data
2  if (myid==0){
3        MPI_Scatterv(myData,myDataLengths,myDataStarts,MPI_INT,
4                     MPI_IN_PLACE,myDataLengths[myid],MPI_INT,0,comm);
5  }
6  else{
7        MPI_Scatterv(myData,myDataLengths,myDataStarts,MPI_INT,
8                     myData,myDataLengths[myid],MPI_INT,0,comm);
9  }
10
11 //local sorting
12 sort(myData,myData+myDataLengths[myid]);
13
14
15 //sampling
16 //local regular sampels are stored in "pivotbuffer"
17 for(int index=0;index<numprocs;index++){
18       pivotbuffer[index]= myData[index*myDataLengths[myid]/numprocs];
   }
```

代码 2 分发待排序数据， 本地数据排序和正则采样

阶段 1 提到的并行算法使这里的 scatter 没有必要，可减少通信。

3.2.3 阶段 3 —— 获得划分主元并广播

进程 0 首先用 MPI_Gather 得到上一阶段的所有样本，之后对这些样本使用快速排序，最后在这些有序的样本中选择（#procs-1）个作为划分主元。

得到划分主元后，进程 0 将其广播给整个通信子。

代码如下：

```
if (myid==0){
      // merged list of samples
      unsigned long tempbuffer[numprocs*numprocs];
      MPI_Gather(MPI_IN_PLACE,numprocs,MPI_LONG,
            tempbuffer,numprocs,MPI_LONG,0,comm);
}
else{
      MPI_Gather(pivotbuffer,numprocs,MPI_LONG,
            tempbuffer,numprocs,MPI_LONG,0,comm);
}


//  Root processor multimerges the lists together and then selects
//  final pivot values to broadcast
if (myid == 0){
      //quiksort merged list of samples
      sort(tempbuffer, tempbuffer+numprocs*numprocs);
      // regularly select numprocs-1 of pivot candidates to broadcast
      // as partition pivot values for myData
      for(int i=0; i<numprocs-1; i++){
            pivotbuffer[i] = tempbuffer[(i+1)*numprocs];
      }
}

//process #0 broadcasts partition pivots
MPI_Bcast(pivotbuffer,numprocs-1,MPI_LONG,0,comm);
```

代码 3 获得划分主元并广播

用 MPI_Gather 合并各个进程上的样本并快速排序，排序后正则选出（#procs-1）个划分主元并广播。

3.2.4 阶段 4 —— 根据划分主元划分本地数据

根据上一阶段的划分主元，将各个进程的本地数据划分成 #procs 个划分类。

代码如下：

```
1  //Offset of a partition class
2  int classStart[numprocs];
3  //length of a partition class
4  int classLength[numprocs];
5  int dataindex=0;
6
7
8  //one classindex for one pivot
9  for(int classindex=0; classindex<numprocs-1; classindex++){
10       classStart[classindex] = dataindex;
11       classLength[classindex]=0;
12
13       // as long as dataindex refers to data in the current class
14       while((dataindex< myDataLengths[myid])
15             && (myData[dataindex]<=pivotbuffer[classindex])){
16             classLength[classindex]++;
17             dataindex++;
18       }
19 }
20
21
22 // last partition class
   classStart[numprocs-1] = dataindex;
   classLength[numprocs-1] = myDataLengths[myid] - dataindex;
```

代码 4 根据划分主元划分本地数据

3.2.5 阶段 5 —— 各个进程中属于同一划分类的数据归并排序

经过阶段 4，各个进程中的数据被划分成了 #procs 个划分类。将各个进程的第 i 个划分类中的数据发送到第 i 个进程中，并对得到的合并数据进行归并排序（归并排序的具体实现请查看代码文件）。

代码如下：

```
1        // buffer to hold merged i^th class
2        unsigned long recvbuffer[myDataSize];
3        // length of merged i^th class
4        unsigned long recvLengths[numprocs];
5        unsigned long recvStarts[numprocs];
6
7        for(int iprocessor=0; iprocessor<numprocs; iprocessor++){
8              //before merging
9              //Gather the length of each class in each process
10             MPI_Gather(&classLength[iprocessor], 1, MPI_LONG,
11                   recvLengths,1,MPI_LONG,iprocessor,comm);
12
13             if (myid == iprocessor){
14                   recvStarts[0]=0;
15                   for(int i=1;i<numprocs; i++){
16                         recvStarts[i] = recvStarts[i-1]+recvLengths[i-1];
17                   }
18             }
19
20             MPI_Gatherv(&myData[classStart[iprocessor]],
21                   classLength[iprocessor],MPI_LONG,
22                   recvbuffer,recvLengths,recvStarts,MPI_LONG,iprocessor,comm);
23       }
24
25       int *mmStarts[numprocs]; // array of list starts
26       for(int i=0;i<numprocs;i++){
27             mmStarts[i]=recvbuffer+recvStarts[i];
28       }
29       multimerge(mmStarts,recvLengths,numprocs,myData,myDataSize);
```

代码5 各个进程中属于同一划分类的数据归并排序

3.2.6 阶段 6 —— 合并各划分（本次实验不要求实现这一步）

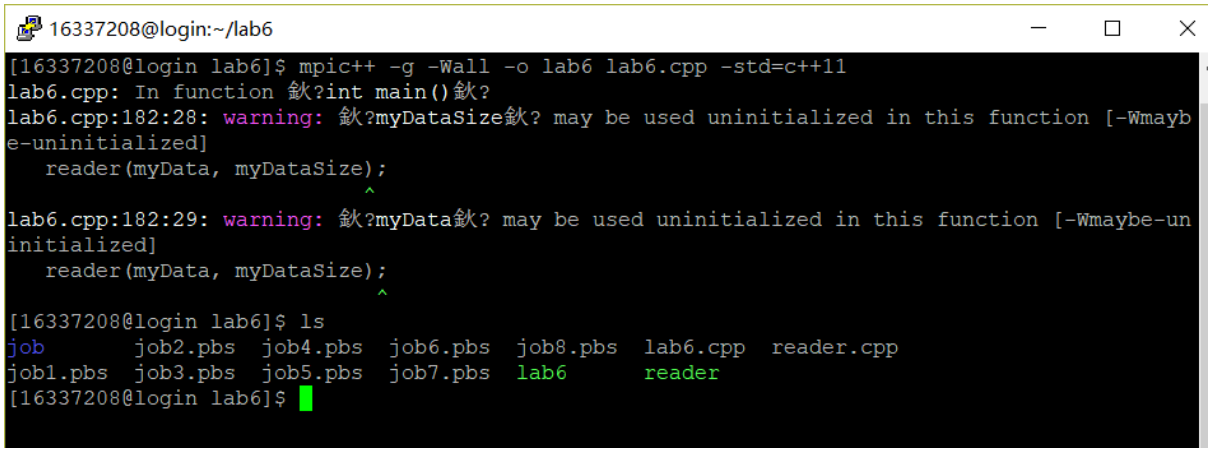
这一阶段将阶段 5 中各个归并排序好后的划分类发送到进程 0 中，备于之后输出等操作。这一步不涉及什么算法，只是简单的进程间通信即可。

由于这一步要求一个进程在内存上申请足够容纳所有数据的空间，对硬件要求很高，这次实验不要求实现这一步。

4 实验过程

4.1 实现正则采样排序算法

用 mpic++成功编译，截图如下：



4.2 提交 PBS 脚本并行运行

我的笔记本内存为 16GB，处理器为 8 核，不适合完成这次实验，所以这次实验我学习使用了 PBS 集群。

我尝试了不同数据大小的运行，发现数据量为$2^{32}, 2^{31}$*或者*2^{30} 的时候都会出现运行错误，输出文件见下图：

```
This job is 3979.login@students
-------------------------------------------------------
Primary job  terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been aborted.
-------------------------------------------------------
-------------------------------------------------------
```

所以我用了较少的数据（2^{26}*个数据*）来运行，具体运行结果见下节。

5 结果分析

5.1 正确性分析

我设计了一个函数来验证一个进程中的数据在算法完成后是单调的。验证的函数如下：

```
string sortedcheck(int xxx[], int xxxStart, int xxxLength){
      for(int i=xxxStart; i<xxxStart+xxxLength-1;i++){
            if (xxx[i]>xxx[i+1])
                  return "NOT Sorted! ";
      }

      return "Sorted! ";
}
```

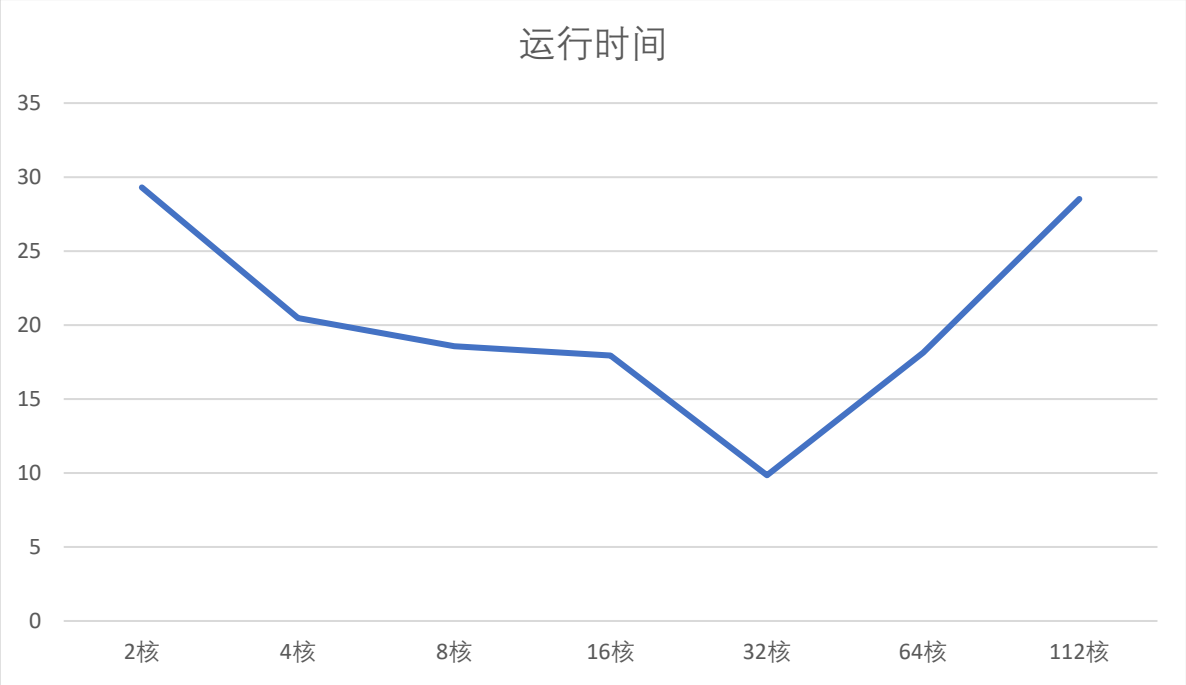
代码 6 算法正确性分析，验证数组有序

在使用大规模的数据实际运行前，我已经使用这个函数确保了自己写的算法确实使各个进程中的数组达到了有序，之后再将数据规模增大。

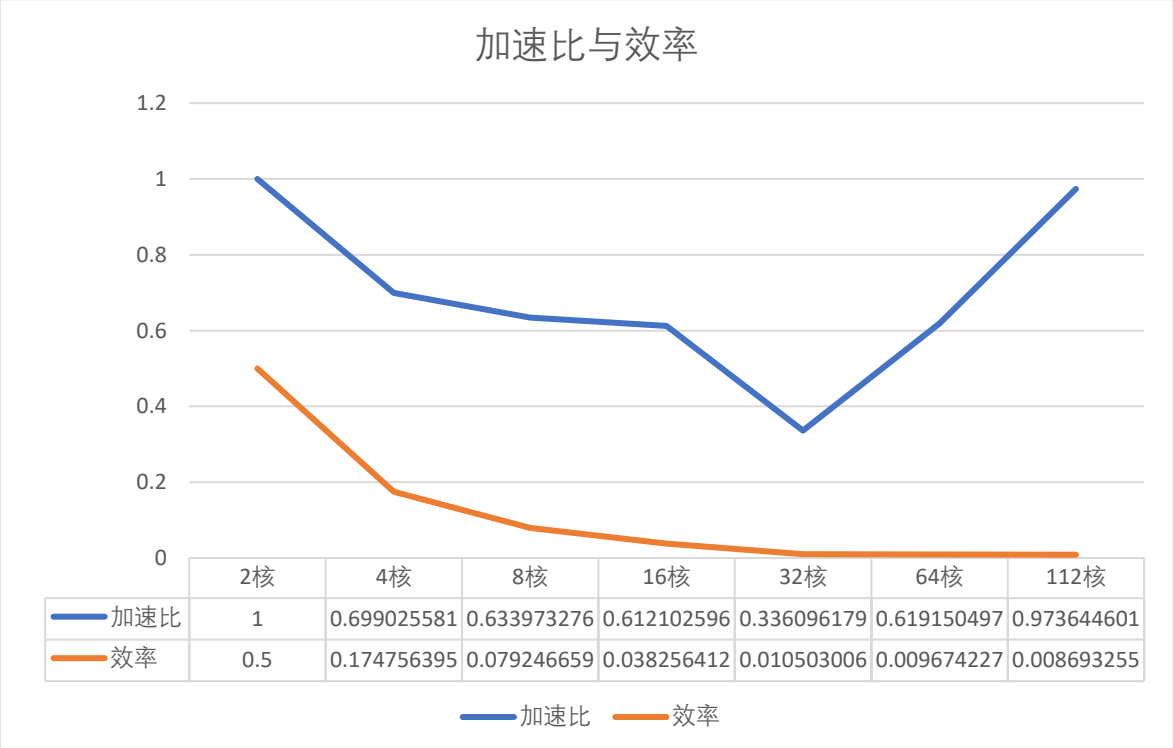
5.2 优化性能分析

n\p	**2 核**	**4 核**	**8 核**	**16 核**	**32 核**	**64 核**	**112 核**
12	0.00113037	0.00048781	0.000650258	0.00139624	0.00364528	0.112481	0.0601209
14	0.00398607	0.00589902	0.0033508	0.00503404	0.00450318	0.135664	0.212978
18	0.078446	0.0576292	0.068093	0.0608235	0.0699674	0.208874	0.316537
22	1.52289	1.07197	1.03313	0.851798	0.526982	1.08644	1.79069
26	29.2995	20.4811	18.5751	17.9343	9.84745	18.1408	28.5273
30	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
31	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
32	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

以数据量为2^{26}的运行结果为例进行分析。不同核数得到的运行用时如下：

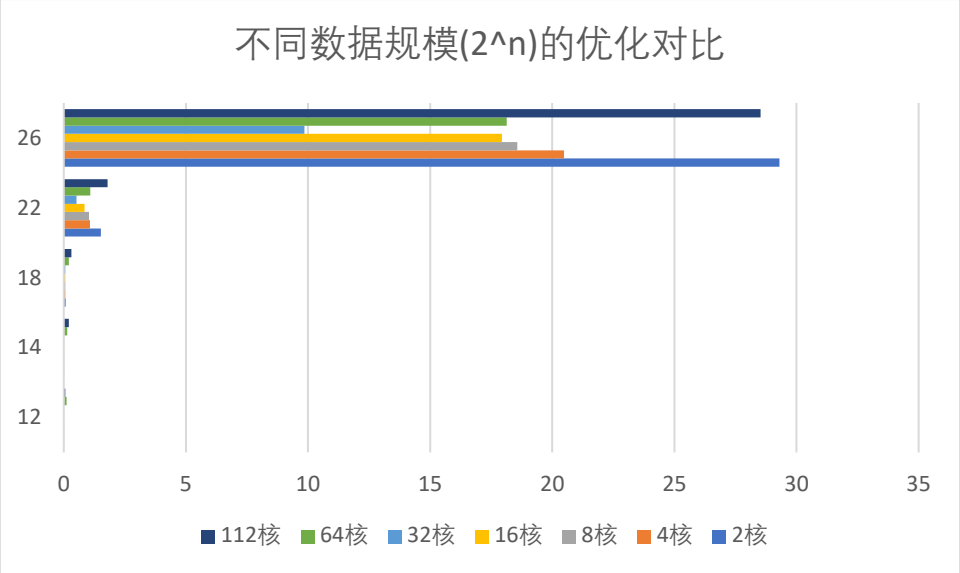


因为到我最终完成这篇报告时，我还没有看到老师公布的串行用时😂所以这里用了 2 进程的用时作为 baseline 计算加速比和效率。



	2核	4核	8核	16核	32核	64核	112核
加速比	1	0.699025581	0.633973276	0.612102596	0.336096179	0.619150497	0.973644601
效率	0.5	0.174756395	0.079246659	0.038256412	0.010503006	0.009674227	0.008693255

对于不同规模的数据的运行结果的优化效果对比如下：



6 反思讨论

6.1 读取文件的方法

如算法部分记录，并行读取似乎是更好的方法，无论是利用 C++文件读取函数实现的还是用 MPI 并行读函数实现的。有必要在接下来的实验尝试和使用这类方法。

6.2 PBS 集群的使用

使用 PBS 集群可以使用到强大得多的计算资源，经过这次实验的熟悉，接下来对硬件要求更高的实验我应使用集群完成。

Work Cited

[1] Li, Xiaobo, et al. "On the versatility of parallel sorting by regular sampling."*parallel computing* 19.10 (1993): 1079-1103.