# 实验 4

学号 PB15000027
姓名 韦清

**实验题目:**
利用 MPI 实现并行排序算法

**实验环境(操作系统,编译器,硬件配置等):**
**操作系统：Ubuntu16.04 （WSL）**
编译器：gcc (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609
硬件配置：Intel® Core™ i7-8650U CPU @ 1.90GHz

**算法设计与分析(写出解题思路和实现步骤)**

PSRS 算法思想如下:

** phase 1: initialization
set up p processors, let the root processor, 0, get data of size n.

** phase 2: scatter data, local sort and regular samples collected
scatter the data values to the p processors. Each processors sorts its local data set, roughly of size n/p, using quicksort.
each processor chooses p sample points, in a very regular manner, from its locally sorted data.

** phase 3: gather and merge samples, choose and broadcast p-1 pivots
the root processor, 0, gathers the p sets of p sample points.
it is important to realize that each set of these p sample points is sorted. These p sets are sorted using multimerge.
from these p^2 sorted points, p-1 pivot values are regularly chosen and are broadcast to the other p-1 processors.

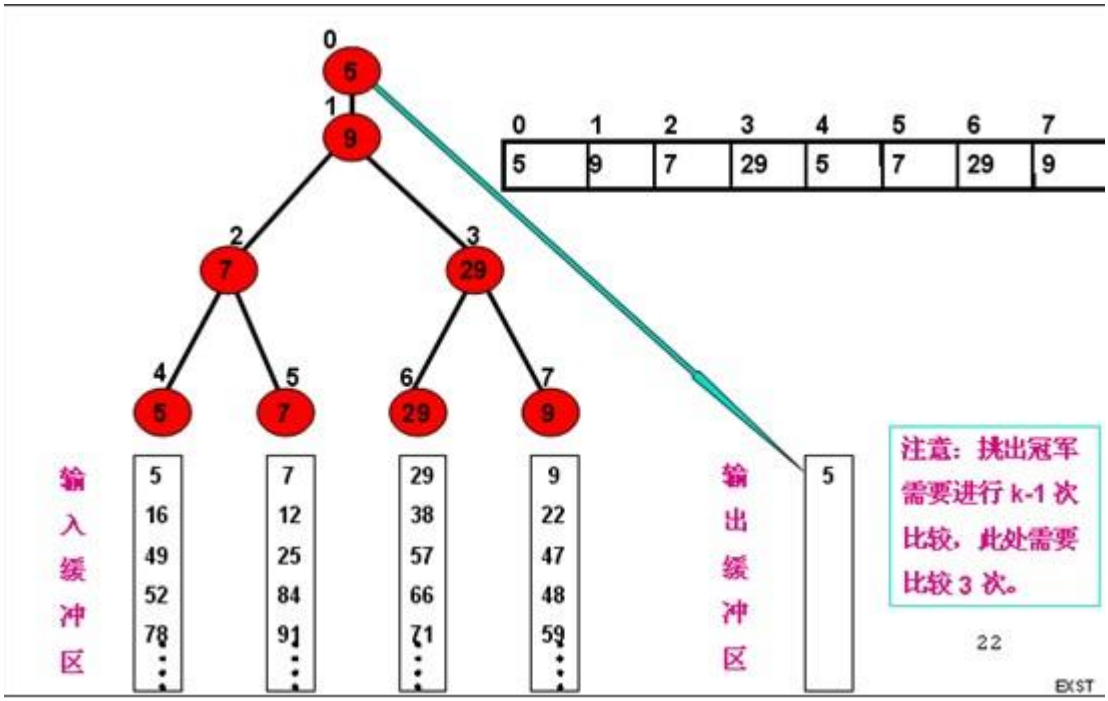
** phase 4: local data is partitioned
each of the p processors partitions its local sorted data, roughly of size n/p, into p classes using the p-1 pivot values.

** phase 5: all *ith* classes are gathered and merged
processor i gathers the ith class of data from every other processor.
each of these classes is sorted using multimerge.

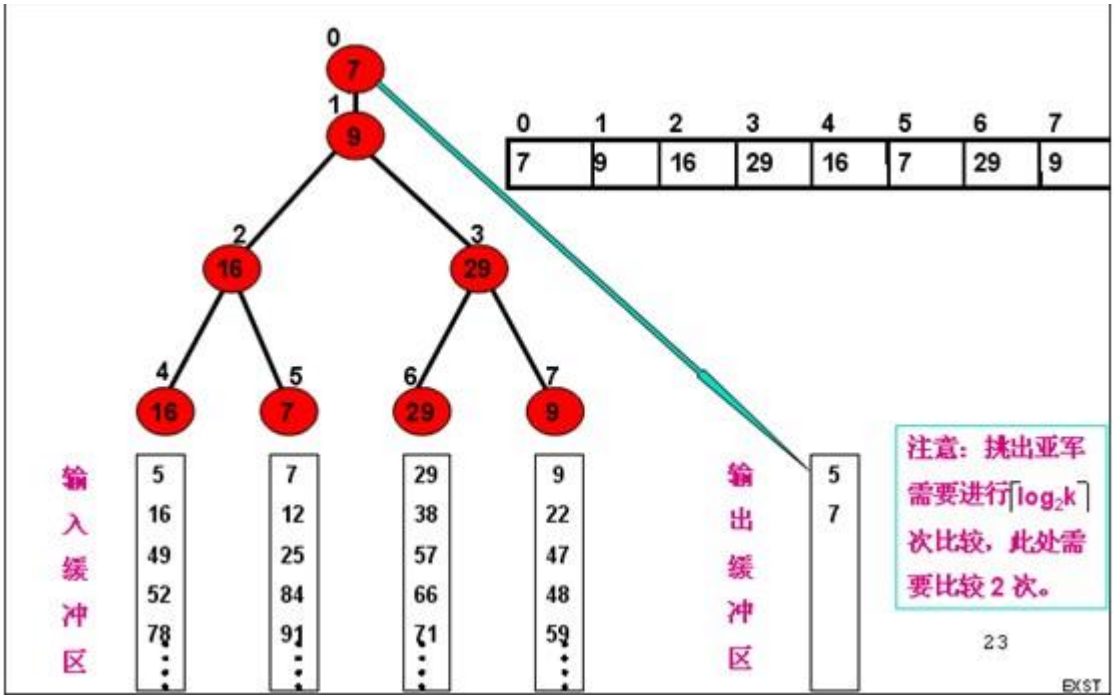
** phase 6: root processor collects all the data
the root processor gathers all the data and assembles the sorted list of n values.

算法思想很简单，难点在于实现多路归并，即上面所说的 MultiMerge 算法。
这里采用败者树的思想进行多路归并：

以四路归并为例：
从树的最底部开始，比较兄弟节点的大小，败者被存入它们的父节点，胜者与它们的叔节点对应的生者进行比较。
以下图为例（图片来源于互联网）
最底部的 5 和 7 比较，7 较大，存在节点 2 的位置，5 较小。
29 和 9 相比，29 较大，将 29 存入节点 3 的位置，9 较小。
对比之前两次的胜者 5 和 9，9 较大，存在节点 1 位置，5 较小，存在节点 0 位置。
5 是这四个输入数组位置 1 元素中最小的一个，将其加入输出 buffer。




因为 5 被输出了，取它对应的输入数组的下一个元素 16.
将 16 和它的父节点的 7 对比，16 较大，将其存入节点 2 位置。
将 7 和节点 2 的父节点的值 9 对比，9 较大，在节点 1 的位置不变，7 较小，将其存入节点 0 位置。
这样一来，可以推出 7 是现在这四个输入数组首元素的最小值，将其输出到输出 buffer，取 7 的下一个元素 12，以此类推。




Multimerge 算法对两路输入数据不适合，2 进程时改成普通的 merge 函数即可。

该算法还有一个难点，是每次的输入数据的大小不是确定的，尤其表现在 phase5 的 multimerge 中，为了解决这个问题，设置一个 MAXKEY，每次数组中的有效数据结束后，在其后加一个 MAXKEY，判断时，即可通过取到 MAXKEY 得知有效数据已经结束。

**核心代码(写出算法实现的关键部分,如核心的循环等)**

```
//phase 1
        if(myid == 0) {
            init(N);
        }
        //phase 2
        MPI_Scatter(data, N/size, MPI_INT, s_data, N/size, MPI_INT, 0,
MPI_COMM_WORLD);
        qsort(s_data, N/size, sizeof(int), my_cmp);
        choose_sample_point(N, size, myid);
        //phase 3
        if(myid == 0) {
            for(int id=1;id<size;id++) {
                MPI_Recv(t[id], size, MPI_INT, id, id, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                t[id][size] = MAXKEY;
            }
            MultiMerge(size, points, size+1);
```

```
            choose_pivot(size);
        }
        else {
            MPI_Send(sample_point, size, MPI_INT, 0, myid, MPI_COMM_WORLD);
        }
        MPI_Bcast(pivot, size-1, MPI_INT, 0, MPI_COMM_WORLD);
        pivot[size-1] = MAXKEY;
        //phase 4
        partition(N, size);
        //phase 5
        for(int id=0;id<size;id++) {
            if(id!=myid) {

MPI_Send(class[id],N/size,MPI_INT,id,(myid+1)*size+id,MPI_COMM_WORLD);
            }
        }
        for(int id=0;id<size;id++) {
            if(id==myid) continue;

MPI_Recv(t[id],N/size,MPI_INT,id,(id+1)*size+myid,MPI_COMM_WORLD,MPI_STATUS_IG
NORE);
        }
        int j;
        for(j=0;j<N/size && class[myid][j]<MAXKEY;j++)
            t[myid][j] = class[myid][j];
        if(j<N/size)
            t[myid][j] = MAXKEY;
        int len = MultiMerge(size, s_data, N/size);
        //phase 6
        if(myid == 0) {
            int count = 0;
            for(j=0;count<N && s_data[j]<MAXKEY;count++,j++) {
                printf("%d\n",s_data[j]);
                data[count] = s_data[j];
            }
            for(int id=1;id<size;id++) {
                MPI_Recv(recv_data[id-
1],N,MPI_INT,id,id+size*size+size,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                for(j=0;count<N && j<N && recv_data[id-1][j]<MAXKEY;j++) {
                    printf("%d\n",recv_data[id-1][j]);
                    data[count++] = recv_data[id-1][j];
                }
            }
            printf("over\n");
```

```
    }
    else {
        MPI_Send(s_data,N,MPI_INT,0,myid+size*size+size,MPI_COMM_WORLD);
    }
```

**实验结果:**

实验结果按如下格式排列

**运行时间**

| 规模\线程数 | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| 1000 | 0.126361847 | 0.66947937 | 0.380992889 | 0.661373138 |
| 10000 | 0.660896301 | 1.313447952 | 1.982688904 | 3.698587418 |
| 1000000 | 86.69042587 | 247.7824688 | 216.7015076 | 223.500967 |
| 3000000 | 256.285429 | 689.7556782 | 522.6712227 | 424.6575832 |

**加速比**

| 规模\线程数 | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| 1000 | 1 | 0.188746439 | 0.331664581 | 0.191059841 |
| 10000 | 1 | 0.50317662 | 0.333333333 | 0.178688842 |
| 1000000 | 1 | 0.34986505 | 0.400045329 | 0.387874948 |
| 3000000 | 1 | 0.371559723 | 0.490337746 | 0.603510779 |

**分析与总结**

**为了方便起见，这里单线程直接调用了 C 语言 stdlib.h 中的库函数 qsort。**

**可以看出，在所选规模下，并行得到的效率并没有串行直接调用 qsort 函数理想。但在同样使用并行程序时，尤其是规模较大时，进程较多时效率是可以明显提高的。**

**原因分析：**

**快速排序函数的期望时间复杂度为 O(nlogn)，由于这里生成输入输入数据时使用了随机生成，所以 qsort 函数很容易就可以达到期望规模。所以并行的排序可能效率并不如串行，尤其是规模较小的情况下。**

**通过分析算法可以看出，PSRS 算法对于任何输入数据都可以取得差不多的时间复杂度，而 qsort 函数受制于输入数据情况，最坏情况下能达到 O(n^2)，所以 PSRS 在很多情况下还是比 qsort 串行效率高的，而且可以达到很高的并行度。**

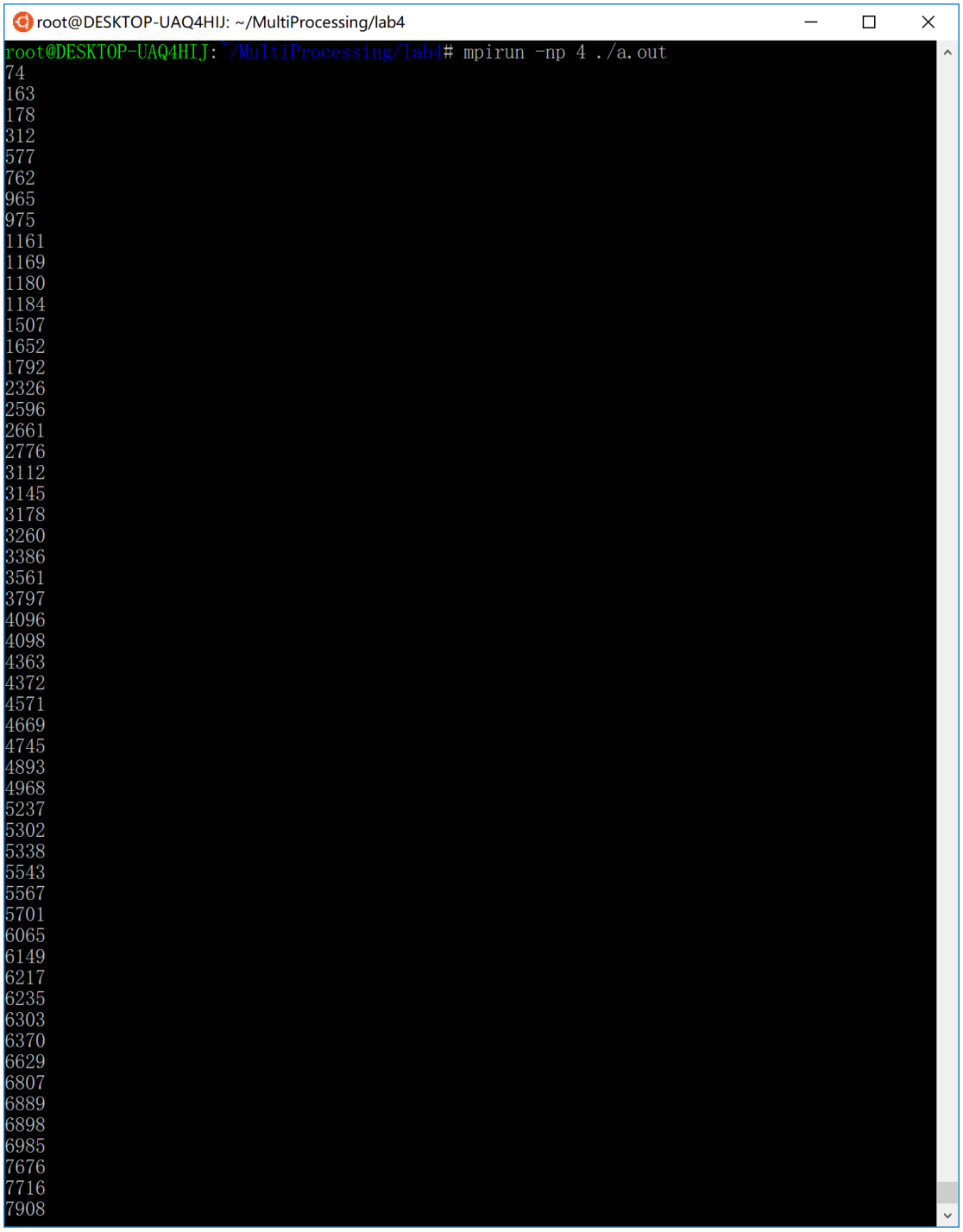
# 部分实验结果截图如下：

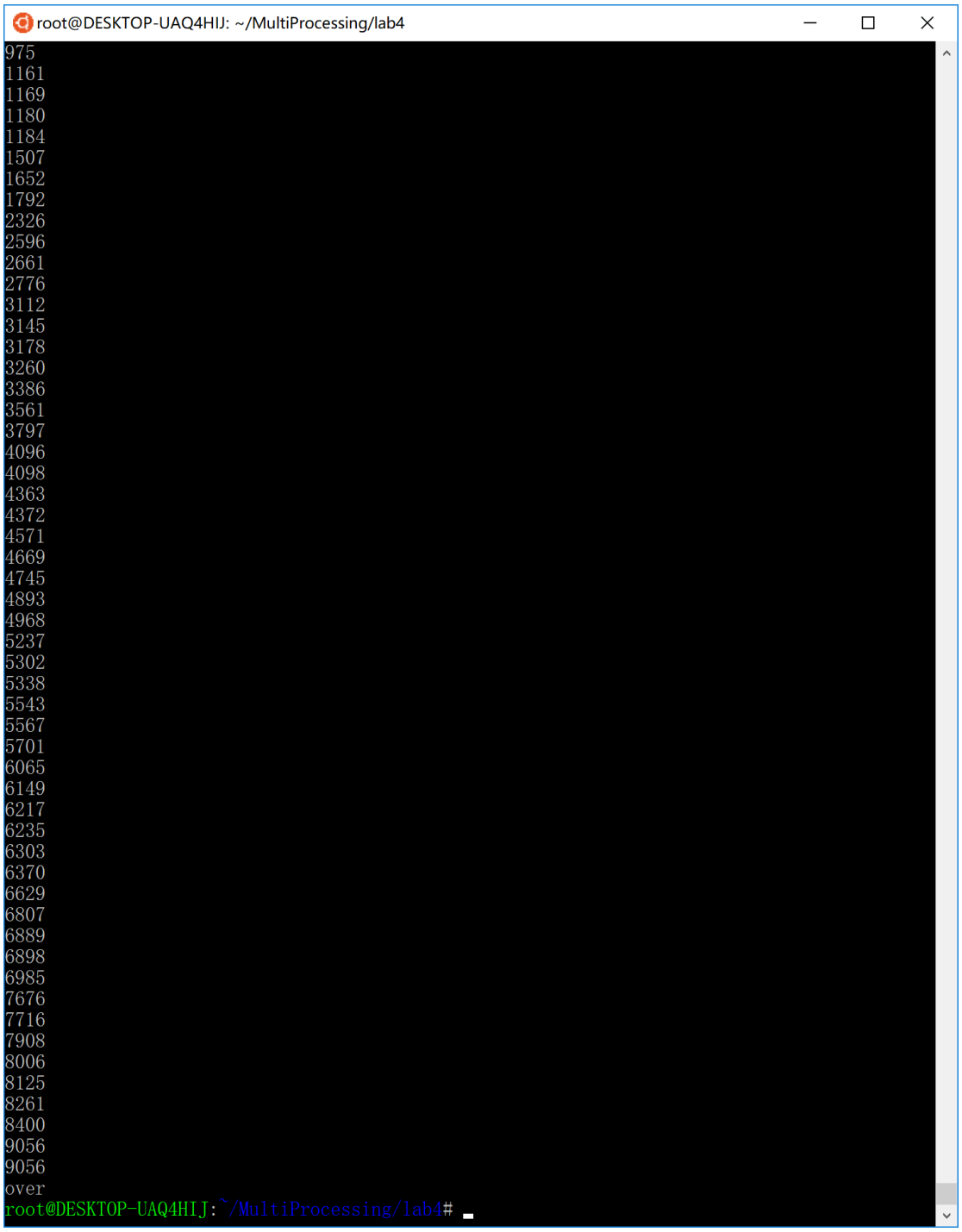
```
root@DESKTOP-UAQ4HIJ:~/MultiProcessing/lab4# mpirun -np 2 ./a.out
begin N = 64
N = 64  time = 0.2939701080

begin N = 1000
N = 1000        time = 0.3786087036

begin N = 10000
N = 10000       time = 2.2482872009

begin N = 30000
N = 30000       time = 52.3440837860
```


root@DESKTOP-UAQ4HIJ: ~/MultiProcessing/lab4
root@DESKTOP-UAQ4HIJ:~/MultiProcessing/lab4# mpirun -np 4 ./a.out
74
163
178
312
577
762
965
975
1161
1169
1180
1184
1507
1652
1792
2326
2596
2661
2776
3112
3145
3178
3260
3386
3561
3797
4096
4098
4363
4372
4571
4669
4745
4893
4968
5237
5302
5338
5543
5567
5701
6065
6149
6217
6235
6303
6370
6629
6807
6889
6898
6985
7676
7716
7908


root@DESKTOP-UAQ4HIJ: ~/MultiProcessing/lab4
975
1161
1169
1180
1184
1507
1652
1792
2326
2596
2661
2776
3112
3145
3178
3260
3386
3561
3797
4096
4098
4363
4372
4571
4669
4745
4893
4968
5237
5302
5338
5543
5567
5701
6065
6149
6217
6235
6303
6370
6629
6807
6889
6898
6985
7676
7716
7908
8006
8125
8261
8400
9056
9056
over
root@DESKTOP-UAQ4HIJ:~/MultiProcessing/lab4#