实验 4

PB17111626 秦亚璇

1 实验题目:

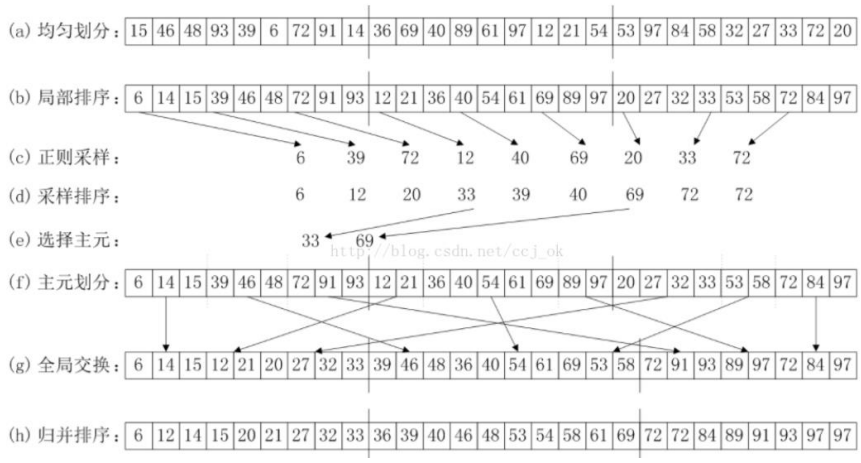
使用 MPI，实现 PSRS（Parallel sorting by regular sampling）算法。

2 实验环境(操作系统,编译器,硬件配置等):

操作系统 Ubuntu 16.04,编译器 g++，硬件配置 ThinkPad X1 Carbon(16G)

3 算法设计与分析(写出解题思路和实现步骤)

这部分的设计主要在于理解 PSRS 算法，下面给出一个举例。



第一步均匀划分明显与之前的并行设计是一致的，我们希望每一个进程能单独完成一部分的局部排序，这里由于 qsort 的调用所以实际情况非常简单，只要按照之前的套路分割数据，排序即可。第三步正则采样其实也没接触过，经查询得到如下的采样 index 公式

$$\lfloor \frac{i \times dataLength}{p} \rfloor,\ i = 0, 1, \ldots p-1$$

很简单地在每个进程上实现一下这个数学公式就可以了，进入下一步采样排序。首先要将所有地 sample 集合起来才能进行有效的排序，这里介绍一个新的 API 叫 MPI_Gather 能够将所有的 sample 汇聚在 root 这里（我所写的程序中是 p0）,此后再在 p0 调用一次 qsort 进行排序即可。

下一步要选择 p – 1 个主元，很简单，index 的公式是 i * p,i = 1…p-1，到此都是在进程 0 上完成的。下面进行主元划分，就是将主元传播到每个已经局部排好序的进程中，然后按照主元将本地的数组分为新的 p 个组，实际上此时共有了 p^2 个已经排好序的数组，按照大小在本地留一个长度为 p 的数组，其中记录了 p 个划分区域的起始 index。

下一步进行全局交换，即将每个进程中的第 i 个分片传送到第 i 个进程中去，所有进程的第 i 分片会在第 i 进程相聚等待下一步的处理，在每个进程上将汇聚的数据进行归并排序，这里的关键问题在于来回的信息传递，存储，和具体 index 的计算，这里运用了 alltoall 和 alltoallv 等关键的 API，最终将所有的排序数组合在一起即为要求的排好序的数组。

4 核心代码(写出算法实现的关键部分,如核心的循环等)

```
double begin, end;
if(bankid == 0)
    begin = MPI_Wtime();//开始计时
int i_begin = bankid * n / numofprocess, i_end = (bankid + 1) * n / numofprocess;
int* localArr = (int*)malloc((i_end - i_begin) * sizeof(int));
for(int i = 0;i < i_end - i_begin;i++)
    localArr[i] = globArr[i + i_begin];
sort(localArr, localArr + i_end - i_begin);//完成分配和局部排序
int* sample = (int*)malloc(numofprocess * sizeof(int));
for (int i = 0; i < numofprocess; ++i)
    sample[i] = localArr[i * (i_end - i_begin) / numofprocess];
int* fullsample = (int*)malloc(numofprocess * numofprocess * sizeof(int));
MPI_Gather(sample, numofprocess, MPI_INT, fullsample, numofprocess, MPI_INT, 0, MPI_COMM_WORLD);
free(sample);//汇集好所有的样本准备进入p-1的挑选工作
    int* privot = new int[numofprocess - 1];
    if (bankid == 0)
    {
        sort(fullsample,fullsample + numofprocess * numofprocess);
        for (int i = 0;i < numofprocess - 1;i++)
            privot[i] = fullsample[(i + 1) * numofprocess];
    }
    free(fullsample);
    MPI_Bcast(privot, numofprocess - 1, MPI_INT, 0, MPI_COMM_WORLD);//传播完了主元
```

```
long long* lowerbound = new long long[numofprocess];
long long* upperbound = new long long[numofprocess];
int* class_begin = (int*)malloc(numofprocess * sizeof(int));
int* class_len = (int*)malloc(numofprocess * sizeof(int));
lowerbound[0] = localArr[0] - (long long)1; class_begin[0] = 0;
upperbound[numofprocess - 1] = localArr[i_end - i_begin - 1];
for (int i = 0; i < numofprocess - 1; ++i)
    lowerbound[i + 1] = upperbound[i] = privot[i];
for (int k = 0, i = 0; k < numofprocess; ++k) {
    while (i < i_end - i_begin && localArr[i] > lowerbound[k] && localArr[i] <= upperbound[k])
        ++i;
    class_len[k] = i - class_begin[k];
    if (k < numofprocess - 1)
        class_begin[k + 1] = i;
}
delete[] lowerbound;
delete[] upperbound;
int* ith_class_len = new int[numofprocess];
for (int i = 0; i < numofprocess; ++i)
    MPI_Gather(class_len + i, 1, MPI_INT, ith_class_len, 1, MPI_INT, i, MPI_COMM_WORLD);
int sumlen = 0;
int* ith_arr_displ = new int[numofprocess];
ith_arr_displ[0] = 0;
for (int i = 0; i < numofprocess; ++i) sumlen += ith_class_len[i];
for (int i = 1; i < numofprocess; ++i) ith_arr_displ[i] = ith_arr_displ[i - 1] + ith_class_len[i - 1];
int* ith_class_arr = new int[sumlen];
for (int i = 0; i < numofprocess; ++i)
    MPI_Gatherv(localArr + class_begin[i], class_len[i], MPI_INT, ith_class_arr, ith_class_len, ith_arr_
```

这些部分之外的部分是做了最初的初始化和最终的合并环节，如需查看，请直接见代码，此处不再占据篇幅了。

5 实验结果

作为验证实验正确性的部分，这里给出 64 个随机数排序的结果，代码最终版本运行时仍可呈现结果。

```
qyxlab@ubuntu:~/Desktop$ mpirun -np 2 ./lab1-2-omp.o
time:0.0000370
  13  124  135  163  198  205  261  293  326  498  563  611  706  791  795  909  966 1038 1098 1108 1112 1118 1225 1313 1370 1383 1553 1601 1691 1946 1951 1963 1975 2040 2132 2136 2153 2163 2164 2234 2260
2284 2350 2394 2397 2531 2536 2609 2793 2918 3153 3250 3277 3327 3378 3389 3458 3499 3548 3559 3764 3835 3881 3958 qyxlab@ubuntu:~/Desktop$
```

运行时间

规模\进程数	1	2	4	8
500w	1.22s	0.73s	0.57s	1.56s

1000w	2.70s	1.46s	1.22s	2.88s
10000w	31.24s	16.97s	12.60s	34.10s

加速比

规模\进程数	1	2	4	8
500w	1	1.67	2.14	0.78
1000w	1	1.85	2.21	0.94
10000w	1	1.84	2.48	0.92

6 分析与总结

在多次进行调整之后，本次实验也收获了相对来说还比较理想的数据，与之前几次一样，当并行程度太大的时候，反效果就出现了，之前我们找到的原因都是通信开销占比相对的增大，在这个实验中不只是通信开销，还有归并操作的每次线程循环都造成了很大的时间消耗，4 进程是这样的规模和算法下相对最合理的进程数选择，当规模相对增大时，加速效果会更加明显。

开始实验一直处于 segmentation fault 的状态，我由于长期犯懒，喜欢直接给出数组之类的静态分配，遇到大规模的问题解决的时候必须使用 malloc 和 free 有选择的及时回收内存，不然就会直接爆掉，此为教训。

这是我在科大的最后一个实验了，实验报告的最后一句话留给我自己，感恩相遇，感谢我在科大遇到的所有人，做过的所有事情，谢谢你们绘成了我十八岁起最美好的青春。