PJ-report

姓名: 陈锐林, 学号:21307130148

2023 年 12 月 13 日

一、实验概况

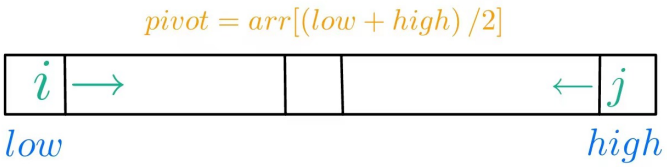
这次 PJ 我选择做 OpenMP 实现并行快速排序和 MPI 实现 Parallel sorting by regular sampling 算法。接下来会分别解释原理、给出串行下的执行时间、解释并行代码和列出不同线程数和数据量下并行的用时。最后会作出总结。

线程数选用 4,8,16；数据量选用 1K，5K，10K，100K。

二、OpenMP 实现并行快速排序

1. 算法原理

常见的快排:如下图,针对索引为 low-high 的元素,取 pivot 为 arr[(low+high)/2]，由两边向中间推动指针 i 和 j，最后再对两侧进行递归快排。



2. 串行用时

串行代码就不展示了，就是下面并行代码去掉 OpenMP 语句的版本。

数据量	1K	5K	10K	100K
用时 (单位:ns)	134500	450800	941600	12886100

3. 并行代码实现

(1) 准备工作：infile 导入之前准备好的一个数组；chrono 库用于提供较高精度的计时

```
int arraySize = 100000;
int arr[arraySize];
int size = arraySize;
ifstream infile("numbers.txt");
int index = 0;
while (infile && index < arraySize) {
    int number;
    infile >> number;
    arr[index++] = number;
}
```

(2) 主函数里设置线程数 +OpenMP 并行的指令

```
#pragma omp parallel num_threads(2)
{
    #pragma omp single nowait
    {
        #pragma omp task
        {
            quicksort(arr, 0, size - 1);
        }
    }
}
```

(3) 在快排里移动指针 i 和 j 之后，进行 OpenMP 并行

```
#pragma omp single nowait
{
    #pragma omp task
    {
        if (left < j)
            quicksort(arr, left, j);
    }

    #pragma omp task
    {
        if (i < right)
            quicksort(arr, i, right);
    }
}
```

4. 并行用时

线程数	数据量			
	1K	5K	10K	100K
4	307900	394300	460000	2178600
8	514000	601000	650600	2624100
16	971000	1259700	1315100	3284000
加速比	0.439	1.15	2.04	5.88

三、MPI 实现 Parallel sorting by regular sampling 算法

1. 算法原理

(1) 均匀分配：根据处理器数 p 划分为 p 段;
(2) 局部排序：每段串行排序;
(3) 正则取样：每段等间距取 p 个样本;
(4) 样本排序：对 p^2 个样本串行排序;
(5) 选择主元：对采样样本取 p-1 个主元;
(6) 主元划分：每段根据主元划为 p 段;
(7) 全局交换：每一段的第 i 小段发送给第 i 号处理器;
(8) 归路排序：每个处理器串行排序.

2. 串行用时

串行即不再使用 MPI 相关接口，直接利用单处理器完成排序。用时如下：能看出，只串行的话有很多没必要的操作；用时比串行快速排序慢多了。

数据量	1K	5K	10K	100K
用时 (单位:ns)	203100	1209500	2184200	28413800

3. 并行代码实现

(1) 调用接口：

```
MPI_Init(&argc, &argv);
auto start = steady_clock::now();
psrs(vec, arraySize);   //排序
auto end = steady_clock::now();
MPI_Finalize();
```

(2) 函数主体：(i) 调用 MPI 函数获取线程 ID 和线程数；并进行该段的排序和取样;[1-3].beg 和 end 标识边界，pivots 存放采样样本

```
MPI_Comm_rank(MPI_COMM_WORLD, &myid);    //获取线程ID
MPI_Comm_size(MPI_COMM_WORLD, &numprocs); //获取总线程数
int num = n / numprocs;
int beg = myid * num;
int end = (myid + 1) * num;
int w = n / (numprocs * numprocs);
end = min(end, n);
int* pivots = new int[numprocs];
quicksort(arr, beg, end, pivots, numprocs, w);
```

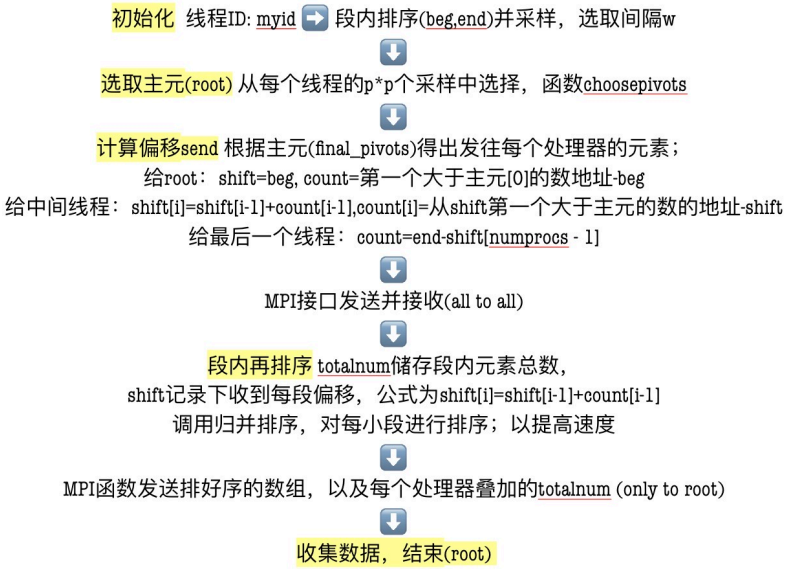
(ii) 调用 MPI_Gather 接收上述取样，排序后选出主元，并用 MPI_Bcast 广播;[4-5]. 根进程进行主元的排序，root_pivots 为 p^2 个采样，final_pivots 为主元

```
int* root_pivots = new int[numprocs * numprocs];
MPI_Gather(pivots, numprocs, MPI_INT, root_pivots, numprocs, MPI_INT, 0, MPI_COMM_WORLD);
if (id == 0)
{
    sort(root_pivots, root_pivots + numprocs * numprocs);
    for (int i = 1; i < numprocs; i++)
        final_pivots[i - 1] = root_pivots[i * numprocs];
}
MPI_Bcast(final_pivots, numprocs - 1, MPI_INT, 0, MPI_COMM_WORLD);
```

(iii) 全局交换并且分段排序，MPI_Alltoallv 完成交换，MergeSort 完成段内排序，MPI_Gather 传给根线程。[7-8]. 其中 receiveNum 是每个进程负责处理的起始位置，recshifts 是偏移

```
MPI_Alltoallv(arr, sendcount, sendshifts, MPI_INT, result, receiveNum, recshifts, MPI_INT, MPI_COMM_WORLD);
int* sort_result = new int[totalnum];
MergeSort(result, sort_result, receiveNum, recshifts, numprocs, totalnum);
int* num_idx = new int[numprocs];
MPI_Gather(&totalnum, 1, MPI_INT, num_idx, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

(iv) 省略了繁琐的 [6]，二次分段和确定偏移等部分。大致的流程图如下：标为 root 的即在根进程才进行，其他是都要做的。



4. 并行用时其中最后一行是对其余的线程数也进行了试验，找到了最适合的线程数；对于 1K、5K 和 10K 的数据量，线程数为 4 就是较好的了。

线程数	数据量			
	1K	5K	10K	100K
4	112300	528200	539500	6055800
8	324100	538800	546900	3656100
16	109968300	120040000	80308000	219884400
10				3174000
加速比	1.82	2.27	4.00	9.09

四、总结

首先，能看出随着数据量的增大，并行的方法的优势会更加明显；在 100K 时都达到最高的加速比。其次是，如果数据量较少，申请过多线程也会让效率低下；OpenMP 中，都只需 4 个线程就达到了最佳情况；而 MPI 中线程申请成本貌似高得多，16 个线程就慢了几个数量级。