实验 4

PB16001715 陈思源

2020 年 7 月 14 日

1 实验题目

利用 MPI 实现并行排序算法。

使用 MPI，实现 PSRS（Parallel sorting by regular sampling）算法，算法具体描述可以参考以下网站：

http://csweb.cs.wfu.edu/bigiron/LittleFEPSRS/build/html/PSRSalgorithm.html

2 实验环境

运行在服务器节点上，操作系统内核 Linux 3.10.0-862.el7.x86_64，使用 gcc 编译器，版本 4.8.5。处理器为 Intel 至强 E5-2620，基准频率 2.00GHz。

3 算法设计与分析

输入：待排序整数的规模 n 和待排序数组 A[1..n]

输出：数组 A[1..n] 经排序后的数组 B[1..n]

资源：p 个进程

解题思路

为了让全局相关的数据可以在多个处理器上并行排序，PSRS 算法将数组预排序为多个局部有序的块，然后处理器之间通过交换数据并二次排序使数据全局有序。

实现步骤

1. 初始化：主处理器获取所有 n 个数据。
2. 分发数据、局部排序、正则采样：主处理器将数据均匀分发给所有处理器，每个处理器对局部的 $\frac{n}{p}$ 个数据排序，然后等间隔地选取 p 个数作为样本。

3. 收集并归并样本、选择 $p-1$ 个主元并广播：主处理器收集所有处理器的样本，经过归并排序后，等间隔地选择中间的 $p-1$ 个数作为主元，然后广播给所有处理器。
4. 局部数据划分：每个处理器根据 $p-1$ 个主元将局部数据划分为 p 段。
5. 全局交换并归并：每个处理器将局部划分好的第 i 段数据发送给第 i 个处理器；每个处理器将受到的 p 个有序数组进行归并排序。
6. 收集数据：主处理器从所有处理器收集完整数组并输出。

4 核心代码

```
// 阶段一
if (rank == 0)
{
    fp = fopen("data.txt", "r");
    fscanf(fp, "%d", &n);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
// 阶段二
recvbuffer = malloc(n * sizeof(int));
mydata = malloc(n * sizeof(int));
if (rank == 0)
{
    for (i = 0; i < n; i++)
    {
        fscanf(fp, "%d", &recvbuffer[i]);
    }
    fclose(fp);
    block_num = n / size;
    local_num = n - (size - 1) * block_num;

    sendcounts = malloc(size * sizeof(int));
    displs = malloc(size * sizeof(int));
    sendcounts[0] = local_num;
    displs[0] = 0;
    for (i = 1; i < size; i++)
    {
        sendcounts[i] = block_num;
        displs[i] = displs[i-1] + sendcounts[i-1];
    }
}
else
{
```

```
33      block_num = n / size;
34      local_num = block_num;
35  }
36  MPI_Scatterv(recvbuffer, sendcounts, displs, MPI_INT, mydata,  local_num,
        MPI_INT, 0, MPI_COMM_WORLD);
37  qsort(mydata, local_num, sizeof(int), cmp);
38  cbuffer = malloc(size * sizeof(int));
39  for (i = 0; i < size; i++)
40  {
41      cbuffer[i] = mydata[i * (local_num / size)];
42  }
43  if (rank == 0)
44  {
45      pivotbuffer = malloc(size * size * sizeof(int));
46      tempbuffer = malloc(size * size * sizeof(int));
47  }
48  // 阶段三
49  MPI_Gather(cbuffer, size, MPI_INT, pivotbuffer, size, MPI_INT, 0,
        MPI_COMM_WORLD);
50  index = malloc(size * sizeof(int));
51  if (rank == 0)
52  {
53      for (i = 0; i < size; i++)
54      {
55          index[i] = i * size;
56      }
57      for (i = 0; i < size * size; i++)
58      {
59          min = 2147483647;
60          minindex = -1;
61          for (j = 0; j < size; j++)
62          {
63              if ((index[j] < (j + 1) * size) && (pivotbuffer[index[j]] < min
                    ))
64              {
65                  min = pivotbuffer[index[j]];
66                  minindex = j;
67              }
68          }
69          tempbuffer[i] = min;
70          index[minindex]++;
71      }
72      for (i = 1; i < size; i++)
73      {
74          cbuffer[i-1] = tempbuffer[i * size];
```

```
75      }
76  }
77  MPI_Bcast(cbuffer, size - 1, MPI_INT, 0,MPI_COMM_WORLD);
78  // 阶段四
79  classStart = malloc(size * sizeof(int));
80  classStart[0] = 0;
81  j = 0;
82  for (i = 0; i < size - 1; i++)
83  {
84      while ((j < local_num) && (mydata[j] < cbuffer[i]))
85      {
86          j++;
87      }
88      classStart[i+1] = j;
89  }
90  classLength = malloc(size * sizeof(int));
91  for (i = 0; i < size - 1; i++)
92  {
93      classLength[i] = classStart[i+1] - classStart[i];
94  }
95  classLength[size - 1] = local_num - classStart[size-1];
96  recvLength = malloc(size * sizeof(int));
97  MPI_Alltoall(classLength, 1, MPI_INT, recvLength, 1, MPI_INT,
        MPI_COMM_WORLD);
98  recvStart = malloc(size * sizeof(int));
99  recvStart[0] = 0;
100 for (i = 1; i < size; i++)
101 {
102     recvStart[i] = recvStart[i - 1] + recvLength[i - 1];
103 }
104 // 阶段五
105 MPI_Alltoallv(mydata, classLength, classStart, MPI_INT, recvbuffer,
        recvLength, recvStart, MPI_INT, MPI_COMM_WORLD);
106 for (i = 0; i < size; i++)
107 {
108     index[i] = recvStart[i];
109 }
110 local_num = recvStart[size - 1] + recvLength[size - 1];
111 for (i = 0; i < local_num; i++)
112 {
113     min = 2147483647;
114     minindex = -1;
115     for (j = 0; j < size; j++)
116     {
117         if ((index[j] < recvStart[j] + recvLength[j]) && (recvbuffer[index[
```

```
            j]] < min))
118         {
119             min = recvbuffer[index[j]];
120             minindex = j;
121         }
122     }
123     mydata[i] = min;
124     index[minindex]++;
125 }
126 // 阶段六
127 MPI_Gather(&local_num, 1, MPI_INT, recvLength, 1, MPI_INT, 0,
        MPI_COMM_WORLD);
128 if (rank == 0)
129 {
130     recvStart[0] = 0;
131     for (i = 1; i < size; i++)
132     {
133         recvStart[i] = recvStart[i - 1] + recvLength[i - 1];
134     }
135 }
136 MPI_Gatherv(mydata, local_num, MPI_INT, recvbuffer, recvLength, recvStart,
        MPI_INT, 0, MPI_COMM_WORLD);
```

Listing 1: N 体问题核心代码

5 实验结果

对每个规模和进程数运行三次并取平均，得到结果如下。

表 1: PSRS 排序的实验结果

(a) 运行时间 (s)

规模 \ 进程数	1	2	4	8
10,000,000	3.331706333	2.652706333	2.291516667	2.225918333
100,000,000	35.774784	26.29415367	20.73250267	18.266068
1,000,000,000	406.091234	252.164461	212.305043	200.69312

(b) 加速比

规模 \ 进程数	1	2	4	8
10,000,000	1	1.2560	1.4539	1.4968
100,000,000	1	1.3606	1.7255	1.9585
1,000,000,000	1	1.6104	1.9128	2.0234

6 分析与总结

理论上，如果 PSRS 算法调用的串行排序算法的时间复杂度为 $O(n\log n)$，那么当 p 远小于 n 时，PSRS 算法的时间复杂度为 $O(\frac{n}{p}\log n)$。在本次实验中，虽然加速比随规模增长而增加，但并行算法加速效果并不明显，在规模达到 10 亿时，8 进程加速比仍然只有 2，说明通信开销很大，或者是实现的归并排序的开销占比较多，通信开销和归并排序的时间复杂度都是 $O(n)$，在 10 亿的规模下，与 $O(\frac{n}{p}\log n)$ 是可比的。