**软件学院2017—2018学年第2学期**

**《并行程序设计》课程大作业**

**作业题目：并行分组**

**学院专业：软件工程**

**年级班级：2015级卓越班班**

**学生姓名：**

**学生学号：**

**指导老师：汤德佑**

**一、问题描述**

分组问题，即将原始数据按照某种标准划分成不同的组别，分组后的的数据称为分组数据。分组问题最常见的应用是SQL查询的Group-By操作。其中，常见的解决方法主要有三种：1、基于归并排序的分组。2、基于分段计数分组法的分组。3、基于哈希表法的分组。传统串行的解决方案已经无法满足日益增大的数据量的需求，因此并行化解决分组问题是目前需要研究的对象，其中对传统方法进行并行化就显得尤为关键。

**二、求解方法**

本次大作业采用之前实现的PSRS算法解决分组问题。分组问题可以用排序的方式进行解决，虽然分组问题并不要求分组后的数据是有序的状态（如哈希实现方式），但是排序方法却能够解决分组问题，因为排好序之后数据相当于已经按照大小值划分好了组别，因此能够解决分组问题。

本次作业首先使用PSRS算法对大规模数据进行排序，由于内存无法装下如此大量的数据，因此只能使用外部排序的方式，每排序好一个顺串就产生一个中间文件，最后对所有数据进行排序后使用归并的操作记录每个数据的分组号以及数据的数量即可。

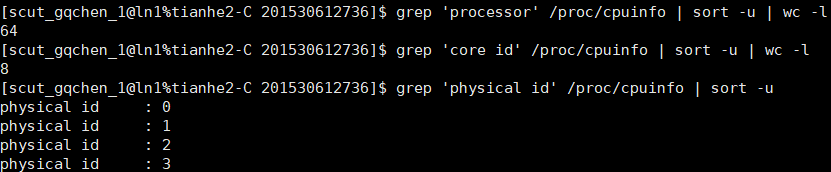
这种求解方案依赖于排序操作，依据相关文献的说明，理论上来说并行化的基于哈希表的分组应该是最快的操作，因为它不要求全局有序，并且只要依据哈希函数找到数据相应的位置即可完成分组。

**三、实验结果截图及性能分析**

1. 实验环境说明：

本次在天河二号上进行的实验，经验证，天河二号节点的运行内存为64G左

右，可以使用的硬盘空间为1000G，由下图所示可以查到天河二号支持的线程总数为64，核心数量为8，物理CPU个数为4个。



(2)采用串行方式实现的运行时间

测试数据集说明：本次使用的数据集为64位的int类型数，数据个数为2G个数，总大小为16GB大小。

在使用单进程、单线程(即假设只有一个主机)，使用排序算法实现分组问题的解法直接对16G大小的数据进行排序后分组，运行时间为401.28 s 。



(3)仅采用MPI实现的性能分析

测试数据集说明：本次使用的数据集为64位的int类型数，数据个数为2G个数，总大小为16GB大小。

数据记录：

单节点上MPI性能测试

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **进程数** | **2** | **4** | **8** | **16** |
| 运行时间(s) | 224.20 | 104.05 | 51.78 | 36.22 |
| 加速比 | 1.78 | 3.85 | 7.74 | 11.07 |
| 效率 | 0.89 | 0.9 | 0.96 | 0.69 |
| **截图编号** | 1 | 2 | 3 | 4 |

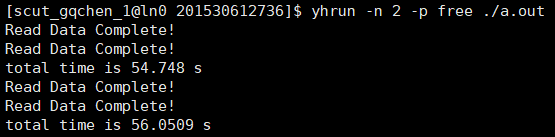


图 1

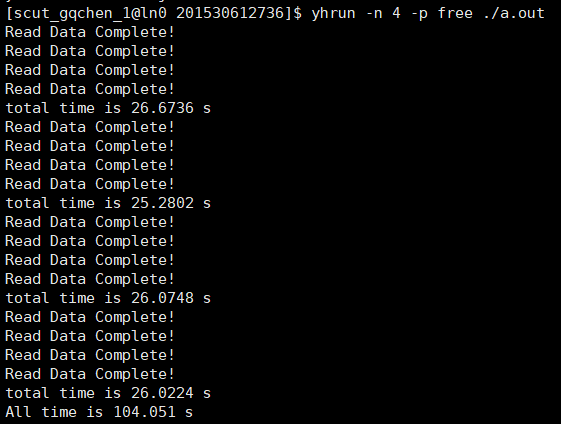


图 2

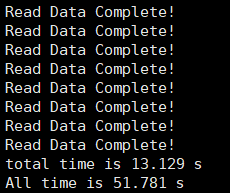


图 3

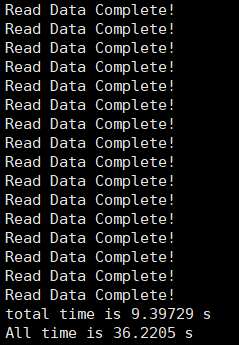
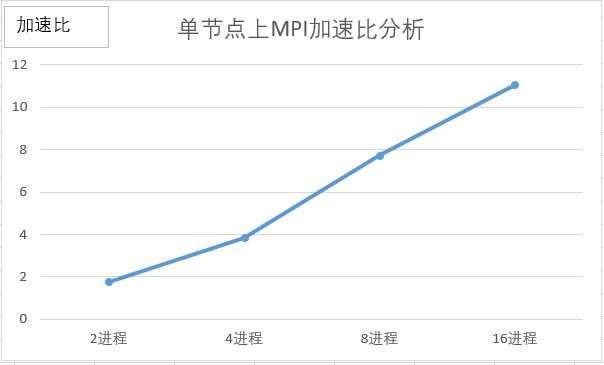


图 4

加速比分析：



弱可拓展性分析：

性能分析：

由实验分析可以得知，天河二号的支持核心数为8个，因此理论上来说8个进程时达到效率最优。由实验可知，当MPI进程数为16时，加速比达到最大值11左右，而MPI进程数为8时，效率达到最大值0.96左右，超过8个进程之后效率开始下降。

(4)采用OpenMP+MPI实现的性能分析

测试数据集说明：本次使用的数据集为64位的int类型数，数据个数为2G个数，总大小为16GB大小。

MPI进程数为2时

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **线程数** | **2** | **4** | **8** | **16** |
| 运行时间(s) | 382.42 | 256.24 | 356.51 | 261.15 |
| 加速比 | 1.04 | 1.56 | 1.12 | 1.53 |
| 效率 | 0.26 | 0.19 | 0.07 | 0.04 |
| **截图编号** | 1 | 5 | 9 | 13 |

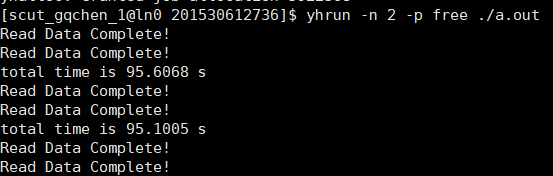


图 1

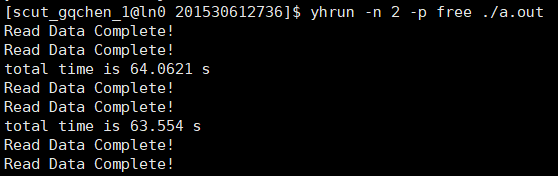


图 5

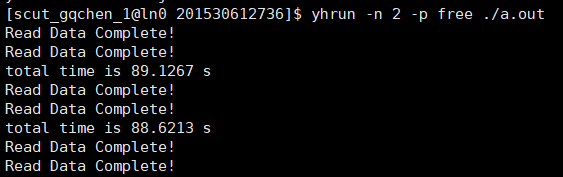


图 9

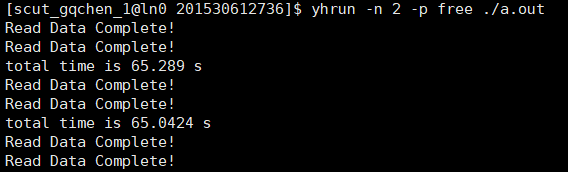


图 11

MPI进程数为4时

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **线程数** | **2** | **4** | **8** | **16** |
| 运行时间(s) | 127.20 | 159.31 | 119.64 | 125.35 |
| 加速比 | 3.15 | 2.51 | 3.35 | 3.2 |
| 效率 | 0.39 | 0.15 | 0.11 | 0.05 |
| **截图编号** | 2 | 6 | 10 | 14 |

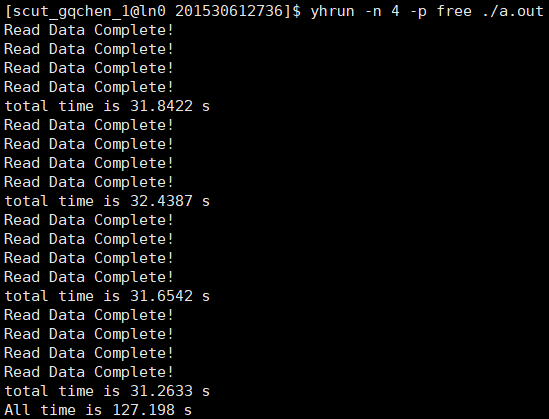


图 2

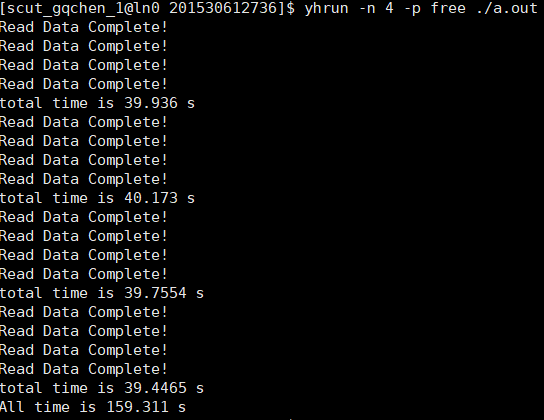


图 6

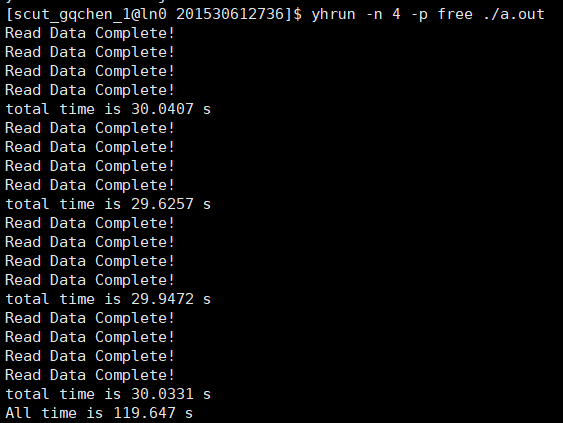


图 10

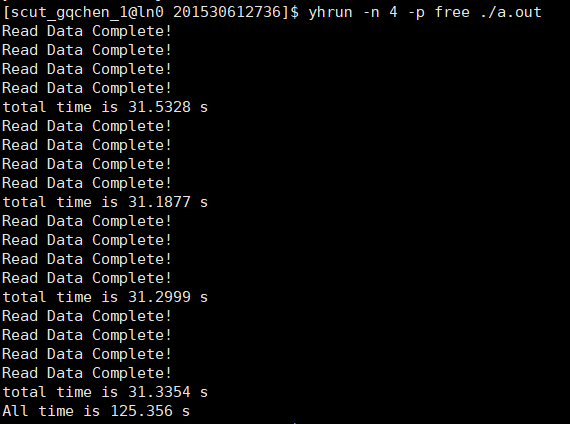


图 14

MPI进程数为8时

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **线程数** | **2** | **4** | **8** | **16** |
| 运行时间(s) | 69.59 | 67.41 | 88.39 | 71.61 |
| 加速比 | 5.76 | 5.95 | 4.53 | 5.61 |
| 效率 | 0.36 | 0.18 | 0.07 | 0.04 |
| **截图编号** | 3 | 7 | 11 | 15 |

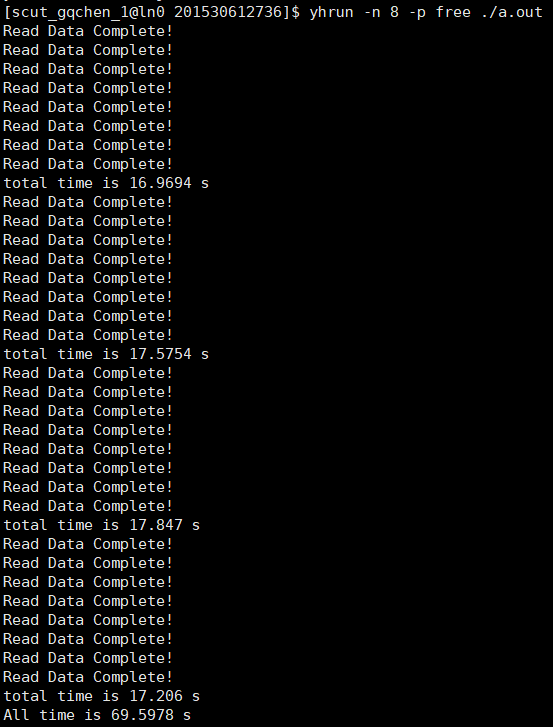


图 3

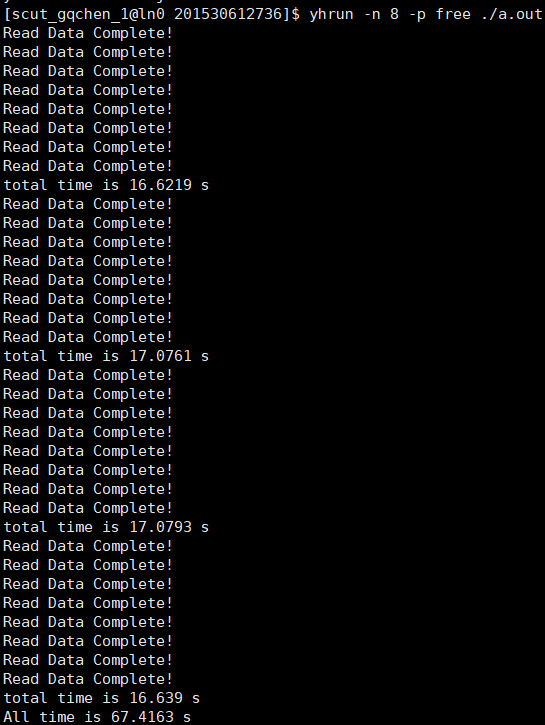


图 7

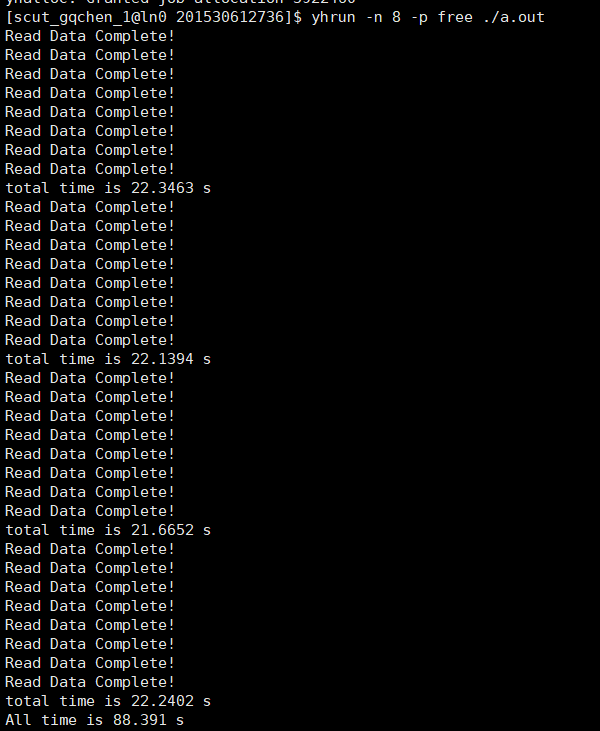


图 11

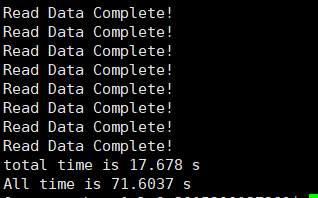


图 15

MPI进程数为16时

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **线程数** | **2** | **4** | **8** | **16** |
| 运行时间(s) | 36.93 | 41.68 | 37.23 | 54.97 |
| 加速比 | 10.86 | 9.62 | 10.77 | 7.32 |
| 效率 | 0.33 | 0.15 | 0.08 | 0.02 |
| **截图编号** | 4 | 8 | 12 | 16 |

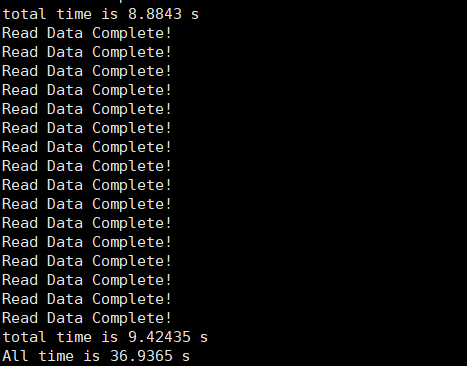


图 4

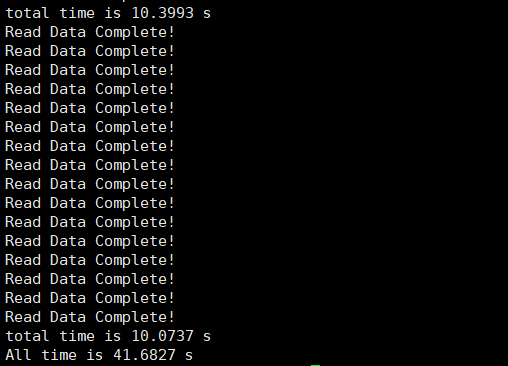


图 8



图 12

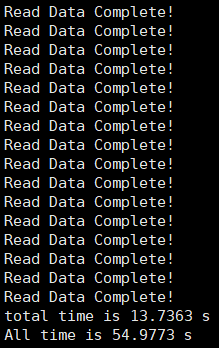
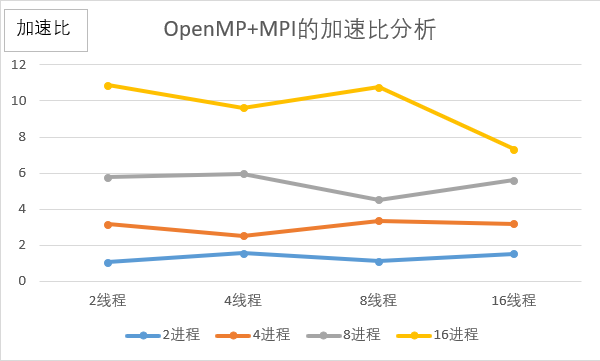
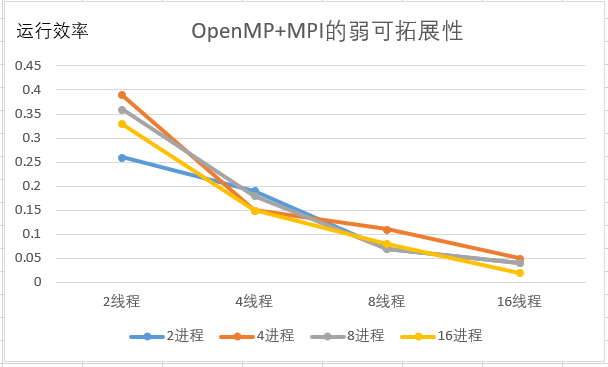


图 16

加速比分析：



弱可拓展性分析：



性能分析：

由实验分析可以得知，天河二号的支持线程数为64个线程，因此理论上来说32个线程时达到性能最优。由实验可知，当MPI进程数为16，线程数为2时，加速比达到最大值10.77左右，在其他分配下同样也是进程数\*线程数的值为32时，加速比达到最大值。实验过程中，效率最大值为4进程2线程的0.39左右，此时加速比为3.2左右；加速比最大值为16进程2线程的10.77左右，效率为0.34左右。

**四、课程总结**

本课程主要对并行程序设计进行了入门。首先课程讲述并行计算产生的背景，然后从软件和硬件两方面大体上描述了如何进行并行计算，接着具体介绍了OpenMP和MPI两种并行化编程的方式。OpenMP是用于共享内存并行系统的多处理器程序设计的一套指导性编译处理方案，提供了对并行算法的高层的抽象描述，程序员通过在源代码中加入专用的pragma来指明自己的意图，由此编译器可以自动将程序进行并行化。与OpenMP并行程序不同，MPI是一种基于信息传递的并行编程技术。消息传递接口是一种编程接口标准，而不是一种具体的编程语言。简而言之，MPI标准定义了一组具有可移植性的编程接口，用于发送和接受消息。最后，课程总结了并行程序开发的整体思路和基本思想，并参观了天河2号超算中心作为课程结束。

本次课程有四次实验课和一次大作业，四次实验课以实现OpenMP+MPI的PSRS算法为核心，实验课以实现OpenMP+MPI的分组问题为核心，均为利用并行化算法和技术来解决实际问题。其中，OpenMP与MPI结合的算法在天河2号上进行运行，实际运行时发现节点经常被大量占用，因此可能实验效果会受到节点占用的影响，假设节点足够空闲，理论上实验的加速比和效率还有可能再进一步提升。

通过这门课我掌握了并行化编程的基本思想，了解了基本的两个并行化技术，对并行化技术进行了实验和大作业的实战，希望将来能有更多的并行化技术应用在相关领域，促进计算机行业的发展。

**五、完整代码**

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <mpi.h>
4. #include<iostream>
5. #include<cstdlib>
6. #include<ctime>
7. #include<vector>
8. #include<fstream>
9. #include<algorithm>
10. #include<math.h>
11. #include <malloc.h>
12. #include<string>
13. #include<cstring>
14. #include<ctime>
15. #include<time.h>
16. #include<omp.h>
17. #include <unistd.h>
18. #include <limits.h>

21. **using** **namespace** std;

24. #define num\_threads 8

27. **void** PSRS(int64\_t \*& array, **long** n)
28. {
30. **int** id;
31. **int** i = 0;
32. **int** base = n / num\_threads;
33. int64\_t p[num\_threads\*num\_threads];
34. int64\_t pivot[num\_threads - 1];
36. **int** yu = n % num\_threads;
37. **int** last\_section\_num = base + yu;

40. **int** \* pivot\_num = **new** **int**[num\_threads];
41. **for** (**int** r = 0; r < num\_threads; r++) {
42. pivot\_num[r] = 0;
43. }
44. int64\_t\*\* pivot2 = **new** int64\_t\*[num\_threads];
45. **int** count[num\_threads] = { 0 };
47. omp\_set\_num\_threads(num\_threads);
49. #pragma omp parallel shared(base,n,i,p,pivot,array,count,pivot2) private(id)
50. {
51. id = omp\_get\_thread\_num();
53. **if** (id != num\_threads - 1)
54. sort(array + id \* base, array + (id + 1)\*base);
55. **else**
56. sort(array + id \* base, array + id \* base + last\_section\_num);
58. #pragma omp critical
59. {
60. **for** (**int** k = 0; k < num\_threads; k++) {
61. p[i++] = array[id \*base + k\*(base / num\_threads + 1)];
62. }
64. }
66. #pragma omp barrier
67. #pragma omp master
68. {
69. sort(p, p + i);
70. **for** (**int** m = 0; m < num\_threads - 1; m++) {
71. pivot[m] = p[(m + 1)\*num\_threads];
72. }
73. }
75. #pragma omp barrier
77. **if** (id != num\_threads - 1)
78. **for** (**int** k = 0; k<base; k++)
79. {
80. **for** (**int** m = 0; m<num\_threads - 1; m++)
81. {
82. **if** (array[id\*base + k] <= pivot[m])
83. {
84. pivot\_num[id] = pivot\_num[id] + 1;
85. **break**;
86. }
88. **if** (array[id\*base + k] > pivot[m] && m == num\_threads - 2) {
89. pivot\_num[id] = pivot\_num[id] + 1;
90. **break**;
91. }
92. }
93. }
94. **else**
95. **for** (**int** k = 0; k<last\_section\_num; k++)
96. {
97. **for** (**int** m = 0; m<num\_threads - 1; m++)
98. {
99. **if** (array[id\*base + k] <= pivot[m])
100. {
101. pivot\_num[id] = pivot\_num[id] + 1;
102. **break**;
103. }
105. **if** (array[id\*base + k] > pivot[m] && m == num\_threads - 2) {
106. pivot\_num[id] = pivot\_num[id] + 1;
107. **break**;
108. }
109. }
110. }
112. #pragma omp barrier
113. pivot2[id] = (int64\_t \*)malloc(pivot\_num[id] \* **sizeof**(int64\_t));
115. #pragma omp barrier
116. **if** (id != num\_threads - 1)
117. **for** (**int** k = 0; k<base; k++)
118. {
119. **for** (**int** m = 0; m<num\_threads - 1; m++)
120. {
121. **if** (array[id\*base + k] <= pivot[m])
122. {
123. pivot2[id][count[id]++] = array[id\*base + k];
124. **break**;
125. }
127. **if** (array[id\*base + k] > pivot[m] && m == num\_threads - 2) {
128. pivot2[id][count[id]++] = array[id\*base + k];
129. }
130. }
131. }
132. **else**
133. **for** (**int** k = 0; k<last\_section\_num; k++)
134. {
135. **for** (**int** m = 0; m<num\_threads - 1; m++)
136. {
137. **if** (array[id\*base + k] <= pivot[m])
138. {
139. pivot2[id][count[id]++] = array[id\*base + k];
140. **break**;
141. }
143. **if** (array[id\*base + k] > pivot[m] && m == num\_threads - 2) {
144. pivot2[id][count[id]++] = array[id\*base + k];
145. }
146. }
147. }

150. }
152. #pragma omp parallel shared(pivot2,pivot\_num,count) private(id)
153. {
154. **int** id = omp\_get\_thread\_num();
156. sort(pivot2[id], pivot2[id] + pivot\_num[id]);

159. **int** start\_position = 0;
160. **for** (**int** i = 0; i < id + 1; i++) {
162. **if** (i == 0)
163. start\_position = 0;
164. **else**
165. start\_position = start\_position + pivot\_num[i - 1];
166. }

169. **for** (**int** i = 0; i < pivot\_num[id]; i++) {
170. array[start\_position + i] = pivot2[id][i];
171. }

174. }
176. #pragma omp master
178. **for** (**int** f = 0; f < num\_threads; f++) {
179. free(pivot2[f]);
180. }
181. **delete**[] pivot2;
183. i = 0;
185. }



190. int64\_t\* ReadData(string fileName, **long** length, **long** offset) {
192. int64\_t \*Array = (int64\_t\*)malloc(length \* **sizeof**(int64\_t));
194. ifstream file;
195. file.open(fileName.c\_str(), ios::in | ios::binary);
197. file.seekg(offset \* **sizeof**(int64\_t));
198. file.read((**char**\*)Array, length \* **sizeof**(int64\_t));
199. **if** (!file.is\_open()) {
200. cout << "open file " << fileName << " fail!" << endl;
201. exit(-1);
202. }
203. file.close();
204. cout << "Read Data Complete!" << endl;
205. **return** Array;
206. }
208. **void** WriteData(int64\_t \* &Array,string fileName, **long** length, **long** offset) {

211. ofstream file;
212. file.open(fileName.c\_str(), ios::out | ios::binary);
214. **if** (!file.is\_open()) {
215. cout << "open file " << fileName << " fail!" << endl;
216. exit(-1);
217. }
219. file.seekp(offset \* **sizeof**(int64\_t));
220. file.write((**char**\*)Array, length \* **sizeof**(int64\_t));
222. file.close();
223. cout << "Write Data Complete!" << endl;
224. }

227. **void** MPI\_PSRS(**int** loops) {
229. **long**  data\_num = pow(2, 20)\*512 ;
230. **double** total\_time = 0;
232. **int** myid, numprocs;
233. MPI\_Init(NULL, NULL);
234. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myid);
235. MPI\_Comm\_size(MPI\_COMM\_WORLD, &numprocs);
237. **for**(**int** loop = 0 ; loop<loops;loop++)
238. {

241. int64\_t \* data = NULL; //store the global data
242. **long** data\_per\_node = data\_num / numprocs;//the number of data for each node to process
244. //store the local data, after partition
245. int64\_t \* local\_data = (int64\_t\*)malloc(data\_per\_node\***sizeof**(int64\_t));
247. //int \* local\_sample = new int[numprocs]; //store the local sample
248. int64\_t \* local\_sample = (int64\_t\*)malloc(numprocs\***sizeof**(int64\_t));
249. **int** count = 0;
251. //int \* root\_sample = new int[numprocs\*numprocs];
252. int64\_t \* root\_sample = (int64\_t\*)malloc(numprocs\*numprocs\***sizeof**(int64\_t));
254. int64\_t \* pivot = **new** int64\_t[numprocs - 1];
256. **int** \* local\_partion\_num = **new** **int**[numprocs];
257. **for** (**int** i = 0; i<numprocs; i++) {
258. local\_partion\_num[i] = 0;
259. }

262. //store the number of data each compute node should process
263. **int**\* local\_partion\_num\_sum = **new** **int**[1];

266. //store the location of each pivot in each compute node
267. **long**\* local\_partition\_location = **new** **long**[numprocs];
268. **for** (**int** i = 0; i<numprocs; i++) {
269. local\_partition\_location[i] = 0;
270. }
271. **long** \* local\_partition\_location\_all = **new** **long**[numprocs\*numprocs];

274. int64\_t \* after\_partition\_data;
276. //read data
277. local\_data = ReadData("../../dazuoye/100G\_File.txt", data\_per\_node, loop\*data\_num + myid\*(data\_num/numprocs));
279. **double** start\_time = MPI\_Wtime();
281. //cout<<"can allocate 2"<<endl;
283. //sort the local data for each node
284. PSRS(local\_data, data\_per\_node);
285. //sort(local\_data,local\_data+data\_per\_node);
287. //sample
288. **for** (**int** i = 0; i<numprocs; i++) {
289. //cout<<"myid "+ to\_string(myid) +" local sample:" + to\_string( local\_data[  i \* (data\_per\_node / (numprocs + 1))    ] ) +"\r\n";
290. local\_sample[i] = local\_data[i \* (data\_per\_node / (numprocs + 1))];
291. }
293. //gather the sample
294. MPI\_Gather(local\_sample, numprocs, MPI\_INT64\_T, root\_sample, numprocs, MPI\_INT64\_T, 0, MPI\_COMM\_WORLD);
296. //sort the sample in the root and pick the pivot
297. **if** (myid == 0) {
299. //sort the local sample
300. sort(root\_sample, root\_sample + numprocs\*numprocs);
302. //select the pivot
303. **for** (**int** i = 0; i<numprocs - 1; i++) {
304. pivot[i] = root\_sample[(i + 1)\*(numprocs + 1)];
305. }
307. }
309. //broadcast the pivot to all compute node
310. MPI\_Bcast(pivot, numprocs - 1, MPI\_INT64\_T, 0, MPI\_COMM\_WORLD);
312. //traverse the local\_data and record the number for each partition
313. **for** (**int** i = 0; i<data\_per\_node; i++) {
315. //record the number for each partition
316. **for** (**int** j = 0; j<numprocs - 1; j++) {
318. **if** (local\_data[i]<pivot[j]) {
319. local\_partion\_num[j] = local\_partion\_num[j] + 1;
320. **break**;
321. }
323. **if** (local\_data[i] >= pivot[j] && j == numprocs - 2) {
324. local\_partion\_num[j + 1] = local\_partion\_num[j + 1] + 1;
325. **break**;
326. }
328. }
330. }
332. int64\_t \*\* local\_partition\_data =  (int64\_t\*\*)malloc(numprocs\***sizeof**(int64\_t\*));
334. //do the copy
335. //1.allocat the space
336. **for** (**int** i = 0; i<numprocs; i++) {
337. local\_partition\_data[i] = (int64\_t\*)malloc(local\_partion\_num[i] \* **sizeof**(int64\_t));
338. }
340. //2.do the copy
341. **int** \* local\_count = **new** **int**[numprocs];
342. **for** (**int** i = 0; i<numprocs; i++) {
343. local\_count[i] = 0;
344. }
346. **for** (**int** i = 0; i<data\_per\_node; i++) {
348. **for** (**int** j = 0; j<numprocs - 1; j++) {
350. **if** (local\_data[i]<pivot[j]) {
351. local\_partition\_data[j][local\_count[j]++] = local\_data[i];
352. **break**;
353. }
355. **if** (local\_data[i] >= pivot[j] && j == numprocs - 2) {
356. local\_partition\_data[j + 1][local\_count[j + 1]++] = local\_data[i];
357. **break**;
358. }
360. }
362. }
364. free(local\_data);
366. //cout<<"can allocate 5.5"<<endl;
367. //reduce the num the each compute node
368. **for** (**int** i = 0; i<numprocs; i++) {
369. MPI\_Reduce(&local\_partion\_num[i], local\_partion\_num\_sum, 1, MPI\_INT, MPI\_SUM, i, MPI\_COMM\_WORLD);
370. }
372. //each compute node store all the partition situation
373. **int** \* local\_partion\_num\_all = (**int**\*)malloc(**sizeof**(**int**)\*numprocs\*numprocs);

376. MPI\_Allgather(local\_partion\_num,numprocs,MPI\_INT,local\_partion\_num\_all,numprocs,MPI\_INT,MPI\_COMM\_WORLD);

379. //allocat the space for the local partition data
380. after\_partition\_data = (int64\_t\*)malloc(**sizeof**(int64\_t)\*local\_partion\_num\_sum[0]);
382. **int** \* reccount =  (**int**\*)malloc(**sizeof**(**int**)\*numprocs);          //new int[numprocs];
383. **int** \* displs =  (**int**\*)malloc(**sizeof**(**int**)\*numprocs);                //new int[numprocs];

386. **for** (**int** j = 0; j<numprocs; j++) {
387. reccount[j] = local\_partion\_num\_all[myid + j\*numprocs];
388. }
390. **for** (**int** j = 0; j<numprocs; j++) {
391. **if** (j == 0){
392. displs[j] = 0;
394. }
395. **else**{
396. displs[j] = displs[j - 1] + reccount[j - 1];
398. }
400. }

403. //do the partition
404. **for** (**int** i = 0; i<numprocs; i++) {
405. MPI\_Gatherv(local\_partition\_data[i], local\_partion\_num[i], MPI\_INT64\_T, after\_partition\_data, reccount, displs, MPI\_INT64\_T, i, MPI\_COMM\_WORLD);
406. }

409. //cout<<"can allocate 7.1"<<endl;
410. free(local\_partition\_data);
412. //cout<<"can allocate 7.2"<<endl;
413. PSRS(after\_partition\_data, local\_partion\_num\_sum[0]);
414. //sort(after\_partition\_data,after\_partition\_data+local\_partion\_num\_sum[0]);

417. //calculate the offset    local\_partion\_num\_all
418. **long** offset = 0;
419. **for**(**int** i=0;i<myid;i++){
420. **for**(**int** j=0;j<numprocs;j++){
421. offset = offset + local\_partion\_num\_all[i+j\*numprocs];
422. }
423. }

426. WriteData(after\_partition\_data,to\_string((**long** **long**)loop)+".txt", local\_partion\_num\_sum[0], offset); //"../../dazuoye/100G\_File.txt", data\_per\_node, myid\*(data\_num/numprocs)

429. ofstream resultfile;
430. resultfile.open((to\_string( ((**long** **long**)loop))+"result"+to\_string( ((**long** **long**)myid))).c\_str(), ios::out | ios::binary);
431. int64\_t finalcount = 1;
432. //iterate the after partition data
433. **for**(**int** i=0;i<local\_partion\_num\_sum[0];i++){
435. **if**(after\_partition\_data[i] != after\_partition\_data[i+1]){
436. resultfile<<after\_partition\_data[i]<<finalcount;
437. finalcount = 1;
438. }**else**{
439. finalcount++;
440. }
441. }

444. //close the resultfile stream
445. resultfile.close();
447. //free all the allocated source
448. free(local\_sample);
449. free(root\_sample);
450. free(local\_partion\_num\_all);
451. free(after\_partition\_data);
452. free(reccount);
453. free(displs);
455. **double** end\_time = MPI\_Wtime();
456. **double** diff\_time = end\_time - start\_time;
458. total\_time = total\_time + diff\_time;
460. MPI\_Barrier(MPI\_COMM\_WORLD);
461. **if** (myid == 0) {
462. cout << "total time is " << diff\_time << " s" << endl;
463. }
465. }
467. **if** (myid == 0) {
468. cout << "All time is " << total\_time << " s" << endl;
469. }
471. MPI\_Finalize();
472. }
474. **void** m\_mergesort(**int** loop){

477. //declare the final result file stream
478. ofstream ofile;
479. ofile.open("result.txt",ios::out | ios::binary );
481. //declare the array to store
482. int64\_t\* data = **new** int64\_t[loop];
483. int64\_t\* read\_buff = **new** int64\_t[1];
485. //declare the array to store the number of each strand
486. **long**\* num\_count = **new** **long**[loop];
487. **for**(**int** i=0;i<loop;i++){
488. num\_count[i] = 0;
489. }
491. //declare the var to store all the sum of the data num
492. **long** sum\_num = 0;
494. //declare the input file stream
495. ifstream\*  infile = **new** ifstream[loop];
496. //open the file stream
497. **for**(**int** i=0;i<loop;i++){
499. infile[i].open( to\_string( ((**long** **long**)i) )+".txt" , ios::in | ios::binary);
501. //seekp move the pointer to the right place
502. infile[i].seekg(num\_count[i] \* **sizeof**(int64\_t));
504. //read the first data of each strand
505. infile[i].read((**char**\*)read\_buff, 1 \* **sizeof**(int64\_t));
506. data[i] = read\_buff[0];
508. //add the count
509. num\_count[i]++;
510. }
512. int64\_t min;
513. **int** min\_index;
514. int64\_t\* write\_buff = **new** int64\_t[1];
515. //begin the loop
516. **while**(sum\_num != pow(2, 20)\*loop){
518. //find the min value in data array
519. min = data[0];
520. min\_index = 0;
521. **for**(**int** i=0;i<loop;i++){
522. **if**(data[i]<min){
523. min = data[i];
524. min\_index = i;
525. }
526. }
527. write\_buff[0] = min;
529. //write the min value into the result file
530. ofile.seekp(sum\_num \* **sizeof**(int64\_t));
532. ofile.write((**char**\*)write\_buff, 1 \* **sizeof**(int64\_t));
533. sum\_num++;

536. //if the file is end do not read data anymore
537. **if**(num\_count[min\_index]==pow(2, 10)){
539. data[min\_index] = INT\_MAX;
541. }**else**{
542. //read the data into buffer
543. //seekp move the pointer to the right place
544. infile[min\_index].seekg(num\_count[min\_index] \* **sizeof**(int64\_t));
546. //read the first data of each strand
547. infile[min\_index].read((**char**\*)read\_buff, 1 \* **sizeof**(int64\_t));
548. data[min\_index] = read\_buff[0];
550. //add the count
551. num\_count[min\_index]++;
552. }
554. }
556. //close the file stream
557. **for**(**int** i=0;i<loop;i++){
558. infile[i].close();
559. }
560. ofile.close();
562. }
564. **int** main(){
566. **int** loop = 4;
568. //sort the data and store them
569. MPI\_PSRS(loop);
571. //do the m\_mergesort and calculate the number
572. m\_mergesort(loop);
574. }