

16 级计科 7 班: 高性能计算 # 实验 4

Due on Tuesday, November 20, 2018

张永东 周二 1-2 节

颜彬

16337269

Content

	Page
1 算法简介	3
2 算法步骤	3
2.1 均匀划分	3
2.2 局部排序	3
2.3 正则采样	3
2.4 样本排序	3
2.5 选择主元	4
2.6 主元划分	4
2.7 全局交换	4
2.8 归并排序	4
3 关键代码	4
3.1 均匀划分	4
3.2 正则采样	5
3.3 选择主元	5
3.4 按主元划分	6
3.5 全局交换	6
3.6 多路归并排序	7

1 算法简介

并行正则采样排序是一个并行排序算法。在算法运行的任意一个时刻, 每个进程都不需要拥有全部的数据。它可以解决对数据量极大的场景下的排序。

算法的最终结果是

- 进程内, 所有的数据有序
- 进程间, 对任意进程 p_i, p_j s.t. $i < j$, 都有 i 进程内的任意一个数小于 j 进程内的任意一个数。

于是数据被排列成全局有序。

在管理这些有序的数据时, 可以选择一个进程作为主进程, 存储所有其他进程的第一个有序数。以该有序数作为分界线。

2 算法步骤

这里假设初始的全部数据都位于 0 号进程。实际上, 算法也允许初始的数据分成 p 份位于 p 个进程中。算法不需要 p 个进程中每个进程拥有的数据数量相等。

假设 p 代表进程数量, n 代表待排序的数字总数。所有的 $/$ 符号代表下取整的除法。

2.1 均匀划分

0 号进程将所有数据尽可能相等地分成 p 份, 并分发到 p 个进程中。设

$$m = n \bmod p$$

则前 $n-m$ 个进程获得 n/p 个数据, 后 m 个进程获得 $n/p + 1$ 个数据。

2.2 局部排序

由于此时每个进程获得的局部数据是乱序的, 故采用快速排序的方法来进行局部排序。可以证明, 在数据属于随机分布时, 在大部分情况下快速排序都能获得较好的期望复杂度。

2.3 正则采样

每个进程从局部有序的数组中等间距地选取 t 个 pivot。设 d_i 代表 i 号进程拥有的数据的数量, A_i 代表局部有序的数组。则进程 i 选取第 k 个 pivot 的方式为

$$P_i : \text{pivot}_k = A[k \times d_i / p] \quad k \in [0, p)$$

2.4 样本排序

所有进程将样本发送给 0 号进程。每个进程发送的样本数都为 p 。0 号进程对所有接收到的样本作排序。

比较合适的方法是采用多路归并排序, 因为每个子序列都是有序的。但一般而言样本数都很小, 故快速排序也能接受。

2.5 选择主元

0 号进程从有序的样本中选择 $p-1$ 个作为主元。将来 $p-1$ 个主元会作为分割的键, 将局部数组分成 p 份。

此时 0 号进程应有 p^2 个样本。设样本数组为 P 。从中选择 $p-1$ 个样本的方式为

$$P_0 : \text{mainPivot}_k = P[(k+1) \times p] \quad k \in [0, p-1)$$

随后 0 号进程把主元广播到所有的进程。

2.6 主元划分

所有进程按 $p-1$ 个主元, 将局部数组分割成 p 份。

为了便于随后的步骤, 比较合适的方法是遍历一遍数组。在遍历的过程中维护两个信息

- count, 一个数组, 记录着 p 个部分中每个部分的长度
- beg_idx, 一个数组, 记录着 p 个部分中每个部分在原局部数组中的起始位置

事实上, 上述两个数组, 知道其中一个就可以求出另外一个。但为了编程方便, 这两个数组都会被构造和维护。

2.7 全局交换

每个进程已经将自己的局部数组分成 p 份了。在这一步中, 每个进程把第 i 份发给进程 i (i 可能为自己的进程号)。

2.8 归并排序

由于每个进程接收到的子数组都是局部有序的, 故此处采用 p 路归并的方式来做最后的排序。

该步结束后, 所有的进程内的数组就有序了。

3 关键代码

3.1 均匀划分

代码 1 是第一步均匀划分所需要的辅助代码。

由于在第一步中使用 `MPI_Scatterv` 来进行数据的划分, 而该函数需要 `sendcounts` 和 `displs` 两个数组来决定每个划分的长度和其实索引。

代码 1 中的 `scatterv_size` 用来计算 `sendcounts`。`scatterv_dipl` 用来计算 `displs`。这两个函数都返回 `malloc` 分配的堆空间, 故需要后续手动释放内存。

其中划分方法按照小小节 2.1 讲述的方式进行。

代码 1: 均匀划分步骤的部分代码

```

1  int* scatterv_size(ul_t size, int world_size) {
2      /*
3       * return the scatter size used in mpi_scatterv
4       * @param size, length of the array
5       * @param world_size, mpi communicator size
6       * @return ret, int* of length world_size. remember to free it.
7       */
8      int* ret = (int*) malloc(sizeof(int) * world_size);
9      int base_size = size / world_size;
10     int more_size = size \% world_size;
11     assert(world_size > more_size);
12     for (int i = 0; i < world_size - more_size; ++i) {
13         ret[i] = base_size;
14     }
15     for (int i = world_size - more_size; i < world_size; ++i) {
16         ret[i] = base_size + 1;
17     }
18     return ret;
19 }
20
21 int* scatterv_dipl(int* scatterv_array, int world_size) {
22     /* return the displacement of the array
23     * @param scatterv_array, the scatter size.
24     * @world_size, num of process
25     * @return ret, the displacement
26     */
27     int* ret = (int*) malloc(sizeof(int) * world_size);
28     ret[0] = 0;
29     for (int i = 1; i < world_size; ++i) {
30         ret[i] = ret[i-1] + scatterv_array[i-1];
31     }
32     return ret;
33 }

```

3.2 正则采样

进程内部对局部数组正则采样。正则采样的代码如代码 2 所示。

首先计算采样的步长 `step_size` 为该进程的局部数组的长度除以 `p`。再用该步长连续采样 `p` 次。其中代码中的 `world_size` 即为 `p`。

3.3 选择主元

0 号进程在所有进程提供的样本中选择主元，并广播给所有进程。如代码 3 所示，为选择主元的代码。

在该过程中，步长为 `world_size`，即进程的数量。需要以该步长连续采样 `world_size - 1` 次。其中第一次采样在 `world_size` 索引处。

代码 2: 正则采样重要代码展示

```
// step3: choose pivot
2 #define ul_t unsigned long
  ul_t* pivot_array = (ul_t*) malloc(sizeof(ul_t) * world_size);
4  ul_t step_size = scatterv_size_array[my_rank] / world_size;
  for (ul_t i = 0; i < world_size; ++i) {
6      pivot_array[i] = my_array[i * step_size];
  }
```

代码 3: 选择主元重要代码

```
//step5: send main pivot to all process
2 for (int i = 0; i < world_size - 1; ++i) {
    main_pivot[i] = all_pivot[(i + 1) * world_size];
4 }
```

3.4 按主元划分

所有进程接收到 0 号进程提供的 $p-1$ 个主元后, 将局部数组分割成 p 份。如代码 4 所示。

该步的主要难点在于, 为了描述分割后子数组的信息, 需要维护两个数组。

第一个数组 `send_count_array` 描述每个部分的长度, 第二个数组 `sdispls_array` 描述每个部分在局部数组中的起始索引。见小小节 2.6 的描述。

代码 4 的第一个循环用于构造 `send_count_array`。在循环中存在两个索引。 i 从 0 遍历到 `mySize`, 用于索引进程的局部数组。`myidx` 从 0 遍历到 `world_size`, 用于索引所有的主元 `main_pivot`。

在遍历局部数组的过程中, 若当前的局部数组元素 (`my_array[i]`) 小于当前的键 (`main_pivot[myidx]`), 则说明当前元素仍落入第 `myidx` 组中, 于是 `send_count_array[idx]` 自增。

否则, 说明当前元素应该要落入下一个桶内。于是一直递增 `myidx`, 直到找到当前元素适合的桶。

显然数组 `sdispls_array` 可以依赖于 `send_count_array` 完全计算得到。第二个循环的作用是计算前者, 它用“求 `send_count_array` 前 $i-1$ 项的累计和”作为自己第 i 项的值。

3.5 全局交换

全局交换是一个比较困难的操作。由于各个进程根据主元划分后得到的子数组是不规整的, 不同的进程、不同的子数组会有不同的长度, 故这会带来以下的问题。

- 如何确定在全局交换后得到的数组 (`swap_array`) 的长度
- 如何确定各个子数组在 `swap_array` 中的起始索引

事实上, 上述的第二点所需要的数组已经在小小节 3.4 求出来了。重点在于解决第一点, 如何得知全局交换后数组的长度。

代码 4: 各进程按照主元划分局部数组的关键代码

```
for (int i = 0; i < mySize; ++i) {  
2   if (my_array[i] < main_pivot[myidx]) {  
       send_count_array[myidx]++;  
4   } else {  
       // array >= index  
6       while (myidx < world_size && my_array[i] >= main_pivot[myidx]) {  
           myidx++;  
8       }  
       send_count_array[myidx]++;  
10  }  
}  
12 int before_disp_sum = 0;  
for (int i = 0; i < world_size; ++i) {  
14     sdispls_array[i] = before_disp_sum;  
       before_disp_sum += send_count_array[i];  
16 }
```

解决第一点十分关键, 因为本实验所有的空间都需要在运行期动态地分配和回收。不知道数组的长度将无法完成内存的分配。

具体代码见代码 5。代码共可分为四个部分 (见注释)。

1. 将“根据主元划分的子数组的长度”信息 (all_count_array) 通过 alltoall 发送到所有的进程中
2. 计算当前进程应该接受的数字总数
3. 计算其他进程发送的子数组在当前进程的汇总数组中的起始索引
4. 使用 alltoallv 函数得到汇总数组

第一部分, 使用 MPI_Alltoall 函数, 每个进程分享自己的 send_count_array 数组 (即 p 个子数组的长度构成的数组), 并最终得到汇总好的长度数组 all_count_array。all_count_array 数组的长度是进程数 p, 其中的第 i 个元素表示接受来自进程 i 的数的个数。

第二部分, 将 all_count_array 数组求和, 即当前进程所要接受的元素总个数。

第三部分, MPI_Alltoallv 函数需要接受参数 rdispls, 代表接收方收到的各个子数组应该被放置的位置。故第三部分用于求出 rdispls 数组。

第四部分, 调用 MPI_Alltoallv 函数, 作交换操作。在这一步结束后, 每个进程都会获得来自其他进程 (和自己) 的一部分子数组。这一步中用到的所有信息都在前三步中计算到。

3.6 多路归并排序

由于数组中的每个子数组是局部有序的, 故采用多路归并是最快的排序方法。

代码 5: 全局交换关键代码

```
// Part1: firstly all to all the counts in all processes
2 int* all_count_array = (int*) malloc(sizeof(int) * world_size);
  MPI_Alltoall(send_count_array, 1, MPI_INT, all_count_array,
4     1, MPI_INT, MPI_COMM_WORLD);

6 // Part2: get how many numbers self process should get.
  int my_recv_count = 0;
8 for (int i = 0; i < world_size; ++i) {
    my_recv_count += all_count_array[i];
10 }

12 // Part3: used in alltoallv function's rdispls param. so need to accumulate.
  int* rdispls = (int*) malloc(sizeof(int) * world_size);
14 rdispls[0] = 0;
  for (int i = 1; i < world_size; ++i) {
16     rdispls[i] = rdispls[i - 1] + all_count_array[i - 1];
  }
18

// Part4: swap.
20 ul_t* my_swap_array = (ul_t*) malloc(sizeof(ul_t) * my_recv_count);
  MPI_Alltoallv(my_array, send_count_array, sdispls_array, MPI_UNSIGNED_LONG,
22     my_swap_array, all_count_array, rdispls, MPI_UNSIGNED_LONG, MPI_COMM_WORLD);
```

此处借用了 STL 的优先队列，将 p 个数组的头加入到优先队列中。只要优先队列不为空，就取出优先队列的头，取出头元素对应的子数组的第一个元素，并把该数组的下一个元素（若不空）加入到优先队列中。如此反复，进程内的数组即可完成排序。

如代码代码 6 所示。Part1 负责将各个子数组的头压入优先队列中。Part2 负责对优先队列进行出队和入队，最终达到使得进程内数组有序。

代码 6: 多路归并排序

```
// Part1
2 priority_queue<State> pq;
  int idx = 0;
4 for (int i = 0; i < world_size; ++i) {
    if (all_count_array[i] != 0) {
6       pq.push(State(my_swap_array[rdispls[i]], i));
    }
8 }
// Part2
10 ul_t* result = (ul_t*) malloc(sizeof(ul_t) * my_recv_count);
  idx = 0;
12 while (!pq.empty()) {
    State top = pq.top(); pq.pop();
14    ul_t data = top.data;
    int queue_id = top.queue_id;
16    result[idx++] = data;
    queue_beg[queue_id]++;
18    bool notFull = (queue_id != world_size - 1
                    && queue_beg[queue_id] < rdispls[queue_id + 1])
20                  || (queue_id == world_size - 1
                    && queue_beg[queue_id] < my_recv_count);
22    if (notFull) {
        pq.push(State(my_swap_array[queue_beg[queue_id]], queue_id));
24    }
}
```