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

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 号进程将所有的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

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

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

$$\mathrm{P}_0 \ : \ \mathrm{mainPivot}_k = P[(k+1) \times p] \quad k \in [0, p-1)$$

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

```
int* scatterv_size(ul_t size, int world_size) {
    /*
    * return the scatter size used in mpi_scatterv
    * @param size, length of the array
    * @param world_size, mpi communicator size
    * @return ret, int* of length world_size. remember to free it.
    */
    int* ret = (int*) malloc(sizeof(int) * world_size);
    int base_size = size / world_size;
    int more_size = size \% world_size;
    assert(world_size > more_size);
    for (int i = 0; i < world_size - more_size; ++i) {
        ret[i] = base_size;
    }
    for (int i = world_size - more_size; i < world_size; ++i) {
        ret[i] = base_size + 1;
    }
    return ret;
}

int* scatterv_dipl(int* scatterv_array, int world_size) {
    /* return the displacement of the array
    *  @param scatterv_array, the scatter size.
    *  @world_size, num of process
    *  @return ret, the displacement
    */
    int* ret = (int*) malloc(sizeof(int) * world_size);
    ret[0] = 0;
    for (int i = 1; i < world_size; ++i) {
        ret[i] = ret[i-1] + scatterv_array[i-1];
    }
    return ret;
}
```

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
#define ul_t unsigned long
ul_t* pivot_array = (ul_t*) malloc(sizeof(ul_t) * world_size);
ul_t step_size = scatterv_size_array[my_rank] / world_size;
for (ul_t i = 0; i < world_size; ++i) {
    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
for (int i = 0; i < world_size - 1; ++i) {
    main_pivot[i] = all_pivot[(i + 1) * world_size];
}
```

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) {
    if (my_array[i] < main_pivot[myidx]) {
        send_count_array[myidx]++;
    } else {
        // array >= index
        while (myidx < world_size && my_array[i] >= main_pivot[myidx]) {
            myidx++;
        }
        send_count_array[myidx]++;
    }
}
int before_disp_sum = 0;
for (int i = 0; i < world_size; ++i) {
    sdispls_array[i] = before_disp_sum;
    before_disp_sum += send_count_array[i];
}
```

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
int* all_count_array = (int*) malloc(sizeof(int) * world_size);
MPI_Alltoall(send_count_array, 1, MPI_INT, all_count_array,
    1, MPI_INT, MPI_COMM_WORLD);

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

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

// Part4: swap.
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,
    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。

4 其他代码

4.1 读数据代码

读取数据的代码如代码 7所示。

使用 fread 和 fseek 的方式读数据比较优雅。因为所有的数据都按 8 字节连续存储在二进制文件里，只需要将文件里的内容一模一样地复制进内存里，再以 (unsigned long *) 的形式去 “解释”，即可拿到所有的数据。

fseek 的作用是调整文件指针的偏移量。将偏移量设置为 sizeof(ul_t)，即可跳过第一个元素来读取。其中 ul_t 是一个 typedef，是 unsigned long 的同名定义。

代码 6: 多路归并排序

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

代码 7: 读取数据的代码

```
FILE* inputfile;
inputfile = fopen(argv[1], "rb");

fread(&SIZE, sizeof(ul_t), 1, inputfile);

int* scatterv_size_array = scatterv_size(SIZE, world_size);

if (my_rank == 0) {
    fseek(inputfile, sizeof(ul_t), SEEK_SET);
    array = (ul_t*) malloc(sizeof(ul_t) * SIZE);
    fread(array, sizeof(ul_t), SIZE, inputfile);
}
```

5 结果展示

5.1 小规模结果展示

跑了小型数据集，3 个线程的结果如代码 8所示。5 个线程的结果如 代码 9所示。

代码 8: 三线程结果

```
    [0] sorted 1
    [0] sorted 2
    [0] sorted 2
    [0] sorted 3
    [0] sorted 3
    [0] sorted 4
    [0] sorted 4
    [1] sorted 5
    [1] sorted 5
    [1] sorted 7
    [1] sorted 7
    [1] sorted 8
    [1] sorted 8
    [1] sorted 9
    [1] sorted 10
    [1] sorted 10
    [1] sorted 13
    [1] sorted 14
    [2] sorted 15
    [2] sorted 17
    [2] sorted 17
    [2] sorted 18
    [2] sorted 19
    [2] sorted 20
    [2] sorted 20
    [2] sorted 22
    [2] sorted 24
    [2] sorted 25
    [2] sorted 25
    [2] sorted 27
    [2] sorted 27
    [2] sorted 28
    [2] sorted 29
    [2] sorted 34
```

5.2 大数据集结果展示

进程数分别取 2, 4, 8, 16, 32, 64, 112，数量级分别取 12，14，18，22，26，30，31 (2^n)，运行结果如表 1所示。运行环境是天河二号的附属结点。

从上到下看。可以看到，（在数量级足够大时）随着数据的数量上升，程序的运行时间越来越长，且与数量级的增长是等比的。这符合对程序的预期。

从左到右看，随着进程数的增多，在进程数增加一倍时，运行时间大约减少一半，也符合对程序的预期。事实上从表 1上看，运行速度不能提升恰好一倍，这是由于线性加速比是无法达到的，只能尽可能接近。最具特点的例子是，数量级为 2^{30} 的一行中，当线程数从 64 增长到 112 时，运行时间几乎没有变化。这是由于随着线程数变大，线程开销和通讯开销胜过计算开销，占据主导地位。可以预计，如果进一步提升线程数量，运行速度反而会减慢。

代码 9: 5 线程结果

```
    [1] sorted 4
    [0] sorted 1
    [0] sorted 2
    [0] sorted 2
    [0] sorted 3
    [0] sorted 3
    [1] sorted 4
    [1] sorted 5
    [1] sorted 5
    [2] sorted 7
    [2] sorted 7
    [2] sorted 8
    [2] sorted 8
    [2] sorted 9
    [2] sorted 10
    [2] sorted 10
    [3] sorted 13
    [3] sorted 14
    [3] sorted 15
    [3] sorted 15
    [3] sorted 17
    [3] sorted 17
    [3] sorted 18
    [4] sorted 19
    [4] sorted 20
    [4] sorted 20
    [4] sorted 22
    [4] sorted 24
    [4] sorted 25
    [4] sorted 25
    [4] sorted 27
    [4] sorted 27
    [4] sorted 28
    [4] sorted 29
    [4] sorted 34
```

表 1中省去了 1 个线程的时间。1 个线程时，排序算法会退化为快速排序（如果局部排序时采用的是快速排序，见小小节 2.2）。

在数据量为 2^{31}, 且开启 8 线程时，不知道为什么程序会在 step1 处卡死，一直没有输出。奇怪的是，在其他线程数时，输出结果都正常。由于课堂的集群太多人排队，抢不到运行的机会；借用的天河二号附属结点环境复杂，同时也不太便于让我长时间 debug，所以暂时不清楚为什么单单这个数据量会无法得到结果。在表格中，这一项被标为 Null。

表 1: 大数据集程序运行结果展示（单位秒）

数量级	*2* 线程	*4* 线程	*8* 线程	*16* 线程	*32* 线程	*64* 线程	*112* 线程
2^{12}	0.005481	0.005473	0.004228	0.005394	0.006804	0.007681	0.010631
2^{14}	0.006865	0.005411	0.004987	0.005618	0.007157	0.008459	0.010801
2^{18}	0.030332	0.018818	0.013870	0.011391	0.009287	0.011933	0.012236
2^{22}	0.304939	0.152921	0.085436	0.055097	0.042358	0.037910	0.062009
2^{26}	5.747613	3.189985	1.511189	0.893695	0.597348	0.433794	0.410755
2^{30}	107.968516	52.372229	30.937190	16.034783	10.517626	6.960675	6.224264
2^{31}	218.547183	109.004943	Null	33.145929	20.747359	15.419143	11.689263

6 遇到的问题

6.1 free 的问题

这次实验得到的经验是，要注意 malloc 和 free 一一配对，在释放内存后不要再访问它。

这个问题很简单，但在复杂的程序中容易出错。在写完代码后，我为了让内存更 ‘‘节约’’，会尽可能地把所有的 free 操作提前。我可能错误地放置了一些 free 的位置，导致一些内存被提前释放掉了。随后的操作中访问了被 os 回收的内存，出现了一些未定义行为。

具体地表现是，程序运行后有可能结果会出错，但每次运行时出错的原因和错误的位置都不同。在我添加了调试代码后，错误又不出现了，如此反复，很难找到原因。

后来我调整好 free 的位置，让其更 ‘‘保守’’ 一点，程序的结果就一直正常了。

6.2 malloc 失败

在集群上跑时，发现部分例子运行失败，发生报错。如图 1。注意到报错中的 Null buffer pointer, 即空指针错误。

定位到该条语句，如代码 10所示。于是推断空指针错误只有一种可能，就是 malloc 函数返回了空指针。于是推断堆内存不足，无法进一步分配内存，故返回了空指针。

这很可能是由于我把许多 free 指令提到了程序较后的地方，占用了大量本可以提前释放的内存，导致内存不足无法分配。也有可能是前面的步骤中多次分配和释放内存，导致产生内存碎片，无法分配出大内存（这点存疑，因为取决于 malloc 的实现）。

解决方法是，尽早地调用 free，不要占用不必要的内存。

图 1: 运行错误时的报错截图 (请放大来看)

代码 10: 报错代码展示

```
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,
    my_swap_array, all_count_array, rdispls, MPI_UNSIGNED_LONG, MPI_COMM_WORLD);
```