奇偶排序大作业 实验报告

管思源 2021012702

源代码

> 注：与提交的版本略有不同，删去了调试信息并增加了注释，核心功能一致

```
void Worker::sort() {
  /** Your code ... */
  // you can use variables in class Worker: n, nprocs, rank, block_len, data

  /* 以下主要是第一步：进程内排序，也包括了两种特殊情况的提前返回 */
  if (out_of_range) return; // 排除取整导致的 out_of_range 特殊情况
  if (block_len < 80) {
    std::sort(data, data + block_len); // 数据量小时使用 std::sort
  } else {
    radix_sort(data, block_len); // 基数排序，自编函数实现
  }
  if (nprocs == 1) return; // 小数据量单进程运行时直接退出，节省时间

  /* 以下是针对 n = nprocs 边界情况的预处理 */
  // 简单来说就是将每个进程内的数据复制为相同的两份，从而避开这种情况
  float* data_pointer = nullptr;
  int block_size = ceiling(n, nprocs);
  if (n - nprocs == 0) {
    float* data_copy = new float[block_len * 2];
    memcpy(data_copy, data, sizeof(float) * block_len);
    memcpy(data_copy + block_len, data, sizeof(float) * block_len);
    data_pointer = data;
    data = data_copy;
    block_len *= 2;
    block_size *= 2;
  }

  /* 以下定义一些标量，其算法意义在性能优化部分有具体说明 */
  const int first_half = (block_len + 1) / 2; // 与前一个进程比较交换的数据量
  const int second_half = block_size / 2; // 发送给后一个进程比较交换的数据量
  const int iter = block_size % 2 ? nprocs + nprocs / 2 : nprocs; // 排序轮数
  // 可以证明，任意序列在进行了 nprocs 次奇偶交换后一定已完成排序（block_size为偶数时）

  /* 以下定义收发消息的 buffer 和异步通信的 request */
  float* recv_buf = new float[second_half];
  float* send_buf = new float[first_half + second_half];
  MPI_Request request;

  /* 以下是主循环体，每循环一次、相邻进程间就完成了一次奇偶交换 */
  for (int i = 0; i < iter; i++) {
    if (i) MPI_Wait(&request, nullptr);
```

```
    if (!last_rank) {
      // 将本进程后半部分数据发送给后一个进程
      MPI_Isend(data + first_half, second_half, MPI_FLOAT, rank + 1, rank,
MPI_COMM_WORLD, &request);
    }
    if (rank) {
      // 接收前一个进程的后半部分数据
      MPI_Recv(recv_buf, second_half, MPI_FLOAT, rank - 1, rank - 1,
MPI_COMM_WORLD, nullptr);
      // 将前一个进程的后半部分数据与本进程的前半部分数据进行归并
      merge(recv_buf, recv_buf + second_half, data, data + first_half,
send_buf);
      if (!last_rank) MPI_Wait(&request, nullptr);
      // 将归并后更小的那部分数据发送回前一个进程
      MPI_Isend(send_buf, second_half, MPI_FLOAT, rank - 1, rank,
MPI_COMM_WORLD, &request);
    } else {
      // 第 1 处，作用见下
      memcpy(send_buf + second_half, data, sizeof(float) * first_half);
    }
    if (!last_rank) {
      // 接收后一个进程发送回来的数据
      MPI_Recv(recv_buf, second_half, MPI_FLOAT, rank + 1, rank + 1,
MPI_COMM_WORLD, nullptr);
    } else {
      // 第 2 处，作用见下
      memcpy(recv_buf, data + first_half, sizeof(float) * (block_len -
first_half));
    }
    // 将 与前一个进程比较后更大的部分数据 和 后一个进程发送回来更小的部份数据 进行归并，作为
本进程本轮排序后的数据
    // 对于第一个进程，“与前一个进程比较后更大的部分数据”采用本进程的前半部分数据，即第 1 处
    // 对于最后的进程，“后一个进程发送回来更小的部份数据”采用本进程的后半部分数据，即第 2 处
    merge(send_buf + second_half, send_buf + second_half + first_half,
recv_buf, recv_buf + block_len - first_half, data);
  }

  /* 释放 buffer 并归还 request */
  MPI_Wait(&request, nullptr);
  delete[] recv_buf;
  delete[] send_buf;

  /* 以下是针对 n = nprocs 边界情况的后处理 */
  // 包括对得到的 data 隔点取样、恢复 block_len 原值
  if (n - nprocs == 0) {
    float* data_copy = data;
    data = data_pointer;
    block_len /= 2;
    memcpy(data, data_copy, sizeof(float) * block_len);
    delete[] data_copy;
  }
}
```

性能优化方式

本次实验我主要采用了两个性能优化的方法：

1. 基数排序：在数据量大的情况下，使用基数排序而非 `std::sort` 进行第一步进程内排序，时间复杂度为 `O(4n+1024)`，优于后者的 `O(4.87NlgN)`
    - 效果：没有经过仔细的实验对比，不过在数据量为8k时，`std::sort` 用时 `0.521 ms`，而基数排序用时 `0.156 ms`
2. 半进程为单位的奇偶排序：为了避免给出的奇偶排序思路中，每个阶段只有一半的进程有计算任务（因为总要将数据发到一个进程进行处理）这样利用率低的情形，以半个进程（将数据分为前后两部分）为单位进行奇偶排序。偶数阶段即为进程内的排序，而奇数阶段则是前一个进程的后半部分数据和后一个进程的前半部分数据合并进行排序，这样每个阶段的每个进程（除第一个进程）都有排序的计算任务。
    - 说明：如果每个进程的数据量为偶数，则等量充分交换，可以证明经过 `进程数` 个奇数阶段+偶数阶段，排序一定完成；如果每个进程的数据量为奇数，则容易发生“滞留”，在最坏情况下需要 `1.5×进程数` 个奇数阶段+偶数阶段
    - 效果：我一开始就是按这个思路编写的，因此也没有对比...

此外，我对 `Send&Recv` 的异步性和减少 `buffer` 拷贝行为进行了常规优化。

测试结果

进程数	**1×1**	**1×2**	**1×4**	**1×8**	**1×16**	**2×16**
运行时间(ms)	3269.542	2124.103	1488.112	1043.352	814.402	813.950
加速比	1	1.539	2.197	3.134	4.015	4.017