实验一：奇偶排序（odd_even_sort）

计04 何秉翔 2020010944

1. sort 源代码

奇偶排序分为以下三步：

- step 1：进程内排序
- step 2：偶数阶段
- step 3：奇数阶段

其中 step 2 和 step 3 交替进行共 "进程数" 轮。

函数 check_pred：与前一个进程（如果有数据）进行合并排序、重新分配。

函数 check_succ：与后一个进程（如果有数据）进行合并排序、重新分配。

详细注释已在**源码**中给出。

```cpp
#include <algorithm>
#include <cassert>
#include <cstdio>
#include <cstdlib>
#include <mpi.h>

#include "worker.h"

void check_pred(float* data, float* curr_data, float* recv_data, int rank, size_t pred_block_len, size_t block_len, int nprocs_not_oor) {
  MPI_Request req[2];
  float recv_val = -1; // receive the min value from the pred process

  if (rank > 0 && rank < nprocs_not_oor) {
    // send the min value to the pred process, and receive the max value from the succ process
    MPI_Isend(&data[0], 1, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD, &req[0]);
    MPI_Irecv(&recv_val, 1, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD,  &req[1]);
    MPI_Waitall(2, req, MPI_STATUS_IGNORE);

    if (recv_val - data[0] > 1e-15) {
      // reallocate data with the pred process
      MPI_Isend(data, block_len, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD, &req[0]);
      MPI_Irecv(recv_data, pred_block_len, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD, &req[1]);
      memcpy(curr_data, data, block_len * sizeof(float)); // copy from data to curr_data
      MPI_Waitall(2, req, MPI_STATUS_IGNORE);
      // get the last [block_len] floats from <curr_data, recv_data>, store in data
      int i = block_len - 1, j = pred_block_len - 1;
      for(int k = block_len - 1; k >= 0; --k) {
        // assert i and j will not be out of bound (block_len <= pred_block_len)
        if (curr_data[i] <= recv_data[j]) {
          data[k] = recv_data[j--];
        } else {
          data[k] = curr_data[i--];
        }
      }
    }
  }
}

```

```cpp
void check_succ(float* data, float* curr_data, float* recv_data, int rank, size_t succ_block_len, size_t block_len, int nprocs_not_oor) {
  MPI_Request req[2];
  float recv_val = -1; // receive the max value from the pred process

  if (rank < nprocs_not_oor - 1) {
    // send the max value to the succ process, and receive the min value from the pred process
    MPI_Isend(&data[block_len - 1], 1, MPI_FLOAT, rank + 1, 0, MPI_COMM_WORLD, &req[0]);
    MPI_Irecv(&recv_val, 1, MPI_FLOAT, rank + 1, 0, MPI_COMM_WORLD, &req[1]);
    MPI_Waitall(2, req, MPI_STATUS_IGNORE);

    if (data[block_len - 1] - recv_val > 1e-15) {
      // reallocate data with the succ process
      MPI_Isend(data, block_len, MPI_FLOAT, rank + 1, 0, MPI_COMM_WORLD, &req[0]);
      MPI_Irecv(recv_data, succ_block_len, MPI_FLOAT, rank + 1, 0, MPI_COMM_WORLD, &req[1]);
      memcpy(curr_data, data, block_len * sizeof(float)); // copy from data to curr_data
      MPI_Waitall(2, req, MPI_STATUS_IGNORE);
      // get the first [block_len] floats from <curr_data, recv_data>, store in data
      size_t i = 0, j = 0, k = 0;
      while(k < block_len) {
        // block_len >= succ_block_len, consider out of bound !!!
        if (j >= succ_block_len) break;
        if (curr_data[i] <= recv_data[j]) {
          data[k++] = curr_data[i++];
        } else {
          data[k++] = recv_data[j++];
        }
      }
      while(k < block_len) {
        data[k++] = curr_data[i++];
      }
    }
  }
}

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

  // step 1: sort inside the process
  std::sort(data, data + block_len);
  if (out_of_range || (nprocs == 1)) return;

  bool is_even_proc = !(rank & 1); // whether the current process is the even idx

  size_t block_size = ceiling(n, nprocs);
  size_t pred_block_len = 0, succ_block_len = 0;

  // nprocs_not_oor: 25, example -> n = 99, nprocs = 28, block_size = 4,  0~23: 4,  24: 3, 25~27: 0
  int nprocs_not_oor = ceiling(n, block_size); // number of process that not out of bound
  // calculate the block len of the pred process
  if (rank > 0 && rank < nprocs_not_oor) {
    pred_block_len = block_size;
  } else if (rank == nprocs_not_oor) {
    pred_block_len = n - block_size * (rank - 1);
  }
  // calculate the block len of the succ process
  if (rank < nprocs_not_oor - 2) {
```

```
    succ_block_len = block_size;
  } else if (rank == nprocs_not_oor - 2) {
    succ_block_len = n - block_size * (rank + 1);
  }

  float *recv_data = new float[block_size]; // recv_buffer of the adjacent process
  float *curr_data = new float[block_len];  // copy of data

  // sort for nprocs_not_oor iterations
  for (int i = 0; i < nprocs_not_oor; i += 2) {
    if (is_even_proc) { // for even process
      check_succ(data, curr_data, recv_data, rank, succ_block_len, block_len, nprocs_not_oor); // step 2: even_part
      check_pred(data, curr_data, recv_data, rank, pred_block_len, block_len, nprocs_not_oor); // step 3: odd part
    } else { // for odd process
      check_pred(data, curr_data, recv_data, rank, pred_block_len, block_len, nprocs_not_oor); // step 2: even_part
      check_succ(data, curr_data, recv_data, rank, succ_block_len, block_len, nprocs_not_oor); // step 3: odd part
    }
  }
  delete []recv_data;
  delete []curr_data;
}
```

2. 性能优化方式

在该实验中，我**最原始**的实现包括以下几个方面：

- 进程内调用 `std::sort()` 排序。
- 分别在偶数阶段和奇数阶段，进行左右进程对之间的合并排序，并重新分配。
  - **不比较**左进程的最大值和右进程的最小值，一视同仁地进行合并排序。
  - 由其中一个进程向另一个进程发送自己的全部数据，由**接收进程进行排序**，排序后把对应数据发送回去。
  - 右进程的数据个数可能小于左进程，因此需要首先**把自己的数据个数告诉左进程**。
  - 检查是否进行下一轮奇偶排序时，采取从左到右逐个通信的方式来模仿 `Reduce`。

以下部分是我所尝试的优化方式，我们以测例 `data/100000000.dat` 来说明优化效果。

2.1 进程对之间互通数据，不区分哪个是排序进程

即对于左右进程而言，同时向对方发送自己的全部数据，在每个进程内都分别进行数据归并，然后各取所需即可。过程如下所示：

```
Process         0               1
Value   (|   3 5 6 8   | |    1 4 7    |)
Value   (|3 5 6 8 1 4 7| |3 5 6 8 1 4 7|)
Value   (|   1 3 4 5   | |    6 7 8    |)
```

优化前后均需要通信两次，而且两次通信量基本一致，比如优化前是需要通信 $3 \times 2 = 6$ 个数据，优化后需要通信 $3 + 4 = 7$ 个数据，而且只有最后两个进程才会有这样的差别。但是优化后两次通信可以放到一起来进行，靠 `MPI` 内部消息通信的实现来同时进行，并且每个进程只需要归并得到自己所需的那部分数据，不需要 `new` 和 `delete` 更多的空间以及归并更多的数据，而优化前只能排序前一次通信，排序后一次通信。

对于 `data/100000000.dat` 而言，优化效果约为：$100ms$

2.2 对于左右进程对之间的合并排序，首先判断是否需要进行

即首先发送左进程的最大值和右进程的最小值，只有当左进程的最大值大于右进程的最小值时，才进行后续的合并排序、重新分配的过程。这样对于不需要进行合并排序的进程对，可以将通信量从 `block_size` 降低到 1。

对于 `data/100000000.dat` 而言，优化效果约为：$100ms$

2.3 对每个进程，通过计算获得前后进程的数据个数，而不是通过通信获得

仿照 `Worker` 构造函数，计算每个进程的前一个进程（如果有数据）的数据个数，后一个进程（如果有数据）的数据个数：`pred_block_len` 和 `succ_block_len`，因此可以节省关于相邻进程数据个数的通信。

对于 `data/100000000.dat` 而言，优化效果约为：$20ms$

2.4 对于每轮奇偶排序，不进行 `check`，直接排序 `nprocs` 轮

我们发现主要的性能瓶颈在于自己实现的 `reduce`，需要从 0 号进程传播到最后一个进程，消息传递时间为线性时间 `O(n)`，因此先后尝试了以下方法：

1. 从两边向中间传播，即从第 0 号进程和第 $n-1$ 号进程，分别向中传递，在中间进程计算出是否需要进行下一轮迭代，然后再往两边扩散。此方法耗时是原来的一半，但仍然是线性的。
2. 采用树形结构，线程数取 2 的幂次，两两进程匹配配对，作为树的叶节点，然后两两传播，将结果存在一个进程内，再进行两两配对，相当于自底向上建二叉树来传播，到根节点也就是中间节点汇总，再自顶向下传播。此方法时间是 `O(logn)`，但是由于进程数最多 56 个并不多，因此效果不明显。
3. 最后采取的方法也就是最终使用的优化方法，即不进行 `check`，直接排序 "进程数" 轮次，即奇数阶段和偶数阶段加起来不超过 "进程数"。因为我们注意到，对于任何一组待排序的序列，在任意一个进程 p_s 内的某个数据 i，其排序后必定落于某个进程 p_t，这个过程最多进行 "进程数" 轮。

对于 `data/100000000.dat` 而言，优化效果约为：$200ms$

3. 测量时间

对于 `data/100000000.dat`，测试结果如下：（$N \times P$ 表示 N 台机器，每台机器 P 个进程）

$N \times P$	运行时间(`ms`)	相对单进程加速比
1×1	12521.085000	1.0000
1×2	6591.642000	1.8995
1×4	3528.531000	3.5485
1×8	1964.075000	6.3751
1×16	1256.028000	9.9688
2×16	1174.739000	10.659