实验一：奇偶排序（odd_even_sort）

`odd_even_sort.cpp` 中 `sort` 函数的源代码，并对实现进行简要说明（可以以代码注释的形式，或者单独写出）。

```
void Worker::sort() {
  /** Your code ... */
  // you can use variables in class Worker: n, nprocs, rank, block_len, data
  if (out_of_range)
    return;
  if (block_len < 100) {
    std::sort(data, data + block_len);
  } else {
    radixSort(data, block_len);
  }
  if (nprocs == 1)
    return;

  size_t block_size = ceiling(n, nprocs);
  const int len1 = (block_len + 1) / 2;
  const int len2 = block_size / 2;
  const int sz = block_size % 2 ? nprocs + nprocs / 2 : nprocs;
  int left = rank - 1;
  int right = rank + 1;
  float *recvbuf = new float[len2];
  float *sendbuf = new float[len1 + len2];

  for (int i = 0; i < sz; i++) {
    if (!rank) {
      // first rank
      std::memcpy(sendbuf + len2, data, sizeof(float) * len1);
      MPI_Sendrecv(data + len1, len2, MPI_FLOAT, right, rank, recvbuf, len2,
                   MPI_FLOAT, right, right, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    } else if (last_rank) {
      // last rank
      MPI_Recv(recvbuf, len2, MPI_FLOAT, left, left, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);
      mergeArrays(recvbuf, len2, data, len1, sendbuf);
      MPI_Send(sendbuf, len2, MPI_FLOAT, left, rank, MPI_COMM_WORLD);
      std::memcpy(recvbuf, data + len1, sizeof(float) * (block_len - len1));
    } else {
      // middle ranks
      MPI_Sendrecv(data + len1, len2, MPI_FLOAT, right, rank, recvbuf, len2,
                   MPI_FLOAT, left, left, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      mergeArrays(recvbuf, len2, data, len1, sendbuf);
      MPI_Sendrecv(sendbuf, len2, MPI_FLOAT, left, rank, recvbuf, len2,
                   MPI_FLOAT, right, right, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    mergeArrays(sendbuf + len2, len1, recvbuf, block_len - len1, data);
```

```
    }

    delete[] recvbuf;
    delete[] sendbuf;
}
```

在 `sort` 函数以外，我实现了 `radixSort` 和 `mergeArrays` 函数。

`radixSort` 函数实现了一个对浮点数数组进行排序的基数排序，由于浮点数在内存中的二进制表示不满足直接排序的要求，通过位运算（特别是对符号位和大小端的处理）将其转换为可排序的无符号整数格式，然后采用四次按字节（每次 8 位）排序，从最低位开始，最终完成排序。最终实现在需要排序的数较多时性能比 `std::sort` 更好。

`mergeArrays` 函数实现了两个有序数组的合并。

`Worker::sort()` 函数包含以下步骤：首先进行进程内排序，当数据量较小时使用 `std::sort`，否则使用 `radixSort`。接下来进行 `sz` 轮并行奇偶排序通信。若当前为首进程，与右边进程交换。若为尾进程，与左边进程交换。中间进程与左右都进行交换，并进行归并处理。最终将合并的结果写回 `data`。

你所做的性能优化方式（如果有）以及效果。

基数排序

查阅资料发现，使用基数排序替代 `std::sort`，在数据量较大时性能更好。故实现了基数排序。测试基数排序与 `std::sort` 的性能数据如下。测试时均使用单个进程。

数据量	`std::sort` 时间（ms）	基数排序时间（ms）
100	0.006000	0.010000
1000	0.054000	0.038000
10000	0.676000	0.291000
100000	8.187000	2.835000
1000000	98.666000	27.511000
10000000	1136.906000	314.216000
100000000	12380.874000	3265.435000

可以看出，基数排序在数据量较大时性能更好，尤其在数据量达到 1000000 时，基数排序的性能比 `std::sort` 提升了 3 倍多。

MPI 通信通过 `MPI_Sendrecv` 双向并发

使用 `MPI_Sendrecv` 可避免死锁风险，同时进行发送和接收，避免了多次通信的开销。

不过没有进行大规模测试，不太清楚效果如何。

通信优化

没有进行全量数据交换，而是仅交换一半，并只归并边界部分，避免数据拷贝浪费。减少了通信和合并时的数据量，提高整体吞吐效率。

也没有进行大规模测试，粗略观察，性能大概提升了约一倍。

给出在 $1\times1, 1\times2, 1\times4, 1\times8, 1\times16, 2\times16$ 进程（$N\times P$ 表示 N 台机器，每台机器 P 个进程）及元素数量 $n=100000000$ 的情况下 `sort` 函数的运行时间，及相对单进程的加速比。（此测试请使用助教提供的 `100000000.dat`）。

进程数	运行时间（ms）	加速比
1×1	3259.769000	1.0000
1×2	2181.674000	1.4942
1×4	1474.671000	2.2105
1×8	1073.049000	3.0379
1×16	841.952000	3.8717
2×16	824.867000	3.9519