`odd_even_sort.cpp` 中的 `sort()` 函数

```
1  void Worker::sort() {
2      /** Your code ... */
3      // you can use variables in class Worker: n, nprocs, rank,
   block_len, data
4  #define CHECK_BEFORE 10
5  #define IF_START_SORT 11
6  #define SEND_LEN 12
7  #define SEND_DATA 13
8  #define SEND_SORTED 14
9  #define SEND_FLAG 15
10     std::sort(data, data + block_len);
11     bool swapped[2] = { true, true };
12     bool isEven = true;
13     char flag, last;
14     size_t nextBlockLen = 0;
15     if (rank) {
16         MPI_Send(&block_len, 1, MPI_UNSIGNED_LONG, rank - 1,
   SEND_LEN, MPI_COMM_WORLD);
17     }
18     if (!last_rank) {
19         MPI_Recv(&nextBlockLen, 1, MPI_UNSIGNED_LONG, rank + 1,
   SEND_LEN, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
20     }
21     float *nextData = new float[nextBlockLen], *mergeData = new
   float[block_len + nextBlockLen];
22     MPI_Request request;
23     while (swapped[0] || swapped[1]) {
24         bool isFirst = isEven ^ (rank & 1);
25         if (isFirst) {
26             if (!last_rank) {
27                 // First
28                 float secondMin;
29                 MPI_Recv(&secondMin, 1, MPI_FLOAT, rank + 1,
   CHECK_BEFORE, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
30                 if (secondMin >= data[block_len - 1]) {
31                     // No need to sort
```

```
32                      flag = 0;
33                  } else {
34                      flag = 1;
35                  }
36                  MPI_Send(&flag, 1, MPI_CHAR, rank + 1,
IF_START_SORT, MPI_COMM_WORLD);
37              }
38          } else {
39              if (rank) {
40                  // Second
41                  MPI_Send(data, 1, MPI_FLOAT, rank - 1,
CHECK_BEFORE, MPI_COMM_WORLD);
42                  MPI_Recv(&flag, 1, MPI_CHAR, rank - 1,
IF_START_SORT, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
43              }
44          }
45          if (flag) {
46              if (isFirst && !last_rank) {
47                  // Receive and sort
48                  MPI_Recv(nextData, nextBlockLen, MPI_FLOAT,
rank + 1, SEND_DATA, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
49                  mergeSort(data, block_len, nextData,
nextBlockLen, mergeData);
50                  MPI_Isend(nextData, nextBlockLen, MPI_FLOAT,
rank + 1, SEND_DATA, MPI_COMM_WORLD, &request);
51              } else if (!isFirst && rank) {
52                  // Send and sort
53                  MPI_Send(data, block_len, MPI_FLOAT, rank - 1,
SEND_DATA, MPI_COMM_WORLD);
54                  MPI_Irecv(data, block_len, MPI_FLOAT, rank - 1,
SEND_DATA, MPI_COMM_WORLD, &request);
55              }
56          }
57          if (rank == nprocs / 2) {
58              if (rank) {
59                  MPI_Recv(&last, 1, MPI_CHAR, rank - 1,
SEND_SORTED, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
60                  flag |= last;
61              }
```

```
62              if (!last_rank) {
63                  MPI_Recv(&last, 1, MPI_CHAR, rank + 1,
SEND_SORTED, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
64                  flag |= last;
65              }
66              if (rank) {
67                  MPI_Send(&flag, 1, MPI_CHAR, rank - 1,
SEND_FLAG, MPI_COMM_WORLD);
68              }
69              if (!last_rank) {
70                  MPI_Send(&flag, 1, MPI_CHAR, rank + 1,
SEND_FLAG, MPI_COMM_WORLD);
71              }
72          } else if (rank < nprocs / 2) {
73              if (rank) {
74                  MPI_Recv(&last, 1, MPI_CHAR, rank - 1,
SEND_SORTED, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
75                  flag |= last;
76              }
77              MPI_Send(&flag, 1, MPI_CHAR, rank + 1, SEND_SORTED,
MPI_COMM_WORLD);
78              MPI_Recv(&flag, 1, MPI_CHAR, rank + 1, SEND_FLAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
79              if (rank) {
80                  MPI_Send(&flag, 1, MPI_CHAR, rank - 1,
SEND_FLAG, MPI_COMM_WORLD);
81              }
82          }
83          else {
84              if (!last_rank) {
85                  MPI_Recv(&last, 1, MPI_CHAR, rank + 1,
SEND_SORTED, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
86                  flag |= last;
87              }
88              MPI_Send(&flag, 1, MPI_CHAR, rank - 1, SEND_SORTED,
MPI_COMM_WORLD);
89              MPI_Recv(&flag, 1, MPI_CHAR, rank - 1, SEND_FLAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
90              if (!last_rank) {
```

```
                MPI_Send(&flag, 1, MPI_CHAR, rank + 1,
SEND_FLAG, MPI_COMM_WORLD);
            }
        }
        if ((isFirst && !last_rank) || (!isFirst && rank)) {
            MPI_Wait(&request, MPI_STATUS_IGNORE);
        }
        swapped[isEven] = flag;
        isEven ^= 1;
    }
    delete [] nextData;
    delete [] mergeData;
#undef CHECK_BEFORE
#undef IF_START_SORT
#undef SEND_LEN
#undef SEND_DATA
#undef SEND_SORTED
#undef SEND_FLAG
}
```

其中 `mergeSort()` 函数实现如下：

```
static void mergeSort(float *a, size_t len1, float *b, size_t
len2, float *c) {
    size_t i = 0, j = 0, k = 0;
    while (i < len1 && j < len2) {
        if (a[i] < b[j]) c[k++] = a[i++];
        else c[k++] = b[j++];
    }
    while (i < len1) c[k++] = a[i++];
    while (j < len2) c[k++] = b[j++];
    for (i = j = k = 0; i < len1; ++i, ++k)
        a[i] = c[k];
    for (; j < len2; ++j, ++k)
        b[j] = c[k];
}
```

实现思路为：

1. 首先对各个进程内的数据使用 `std::sort()` 排序。
2. 然后开启奇偶排序的循环，对于当前阶段（奇数还是偶数），以及当前进程 `rank`，判断当前属于相邻元素的前一个还是后一个。
3. 对于前一个的情况，接收来自后一个进程的最小数数据，判断是否存在顺序错误，将判断结果发送给后一个进程。

   如果需要排序，则接收来自后一个进程的数据，进行单次归并排序，并将结果的后半发送回后一个进程。
4. 对于后一个的情况，首先发送最小数数据给前一个进程，然后接收来自前一个进程的判断结果。

   如果需要排序，则发送数据给前一个进程，并等待前一个进程的回复。
5. 对各个进程的排序与否进行整合，并确定本次完整的阶段是否存在元素交换，将结果传递给所有进程。
6. 当连续两次均无元素交换时，停止循环。

性能优化与效果

1. 对于 `srun` 指令，测试了 `--cpu-bind` 选项的不同参数，包括 `ldoms`、`cores`、`sockets`、`boards`、`threads`、`quiet`、`verbose`、`rank`，最终选取了 `sockets` 作为参数，平均可以相比默认选项快 $10ms$。
2. 对于前一个进程回传数据的过程，采用了非阻塞模式，大约可以加速 $15ms$ 左右。
3. 在进行下一轮迭代通讯时，采用两侧到中间依次发送，再从中间向两边发散的方式，将两侧的通讯基本做到并行，大约加速在 $10ms$ 左右。

不同进程数运行结果

机器数 $\times$ 进程数	运行时间(ms)	加速比
1×1	12466.713	1.00
1×2	6964.278	1.79
1×4	3985.209	3.13
1×8	2521.986	4.94
1×16	1819.862	6.85
2×16	1263.996	9.86