奇偶排序实验报告

容逸朗　2020010869

源代码

- `odd_even_sort.cpp` 源代码

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

#include "worker.h"

// 合并两个有序序列 src1, src2, 取最大的 len1 个元素写回.
void get_large(float* src1, int len1, float* src2, int len2, float* dst) {
  int pt1 = len1 - 1, pt2 = len2 - 1;
  for (int i = len1 - 1; i >= 0; i--) {
    if (pt2 < 0 || (pt1 >= 0 && src1[pt1] > src2[pt2])) {
      dst[i] = src1[pt1--];
    } else {
      dst[i] = src2[pt2--];
    }
  }
  // 把数据从缓冲区写回 data(src1) 中
  memcpy(src1, dst, len1 * sizeof(float));
}

// 合并两个有序序列 src1, src2, 取最小的 len1 个元素写回.
void get_small(float* src1, int len1, float* src2, int len2, float* dst) {
  int pt1 = 0, pt2 = 0;
  for (int i = 0; i < len1; i++) {
    if (pt2 >= len2 || (pt1 < len1 && src1[pt1] < src2[pt2])) {
      dst[i] = src1[pt1++];
    } else {
      dst[i] = src2[pt2++];
    }
  }
  memcpy(src1, dst, len1 * sizeof(float));
```

```cpp
}

void Worker::sort() {
  // 无元素的进程不参与排序
  if (out_of_range) return;

  // 对当前进程的私有元素排序
  std::sort(data, data + block_len);

  // 只有一个进程的情况下，排序结束
  if (nprocs == 1) return;

  // 计算相邻进程的数据量
  size_t block_size = ceiling(n, nprocs);
  int block_len_left = (int)block_size;
  int block_len_right = (int)std::min(block_size, n - block_size * (rank + 1));

  // 接收数据和排序数据缓冲区
  float *buffer = new float[std::max(block_len_left, block_len_right)];
  float *tmp = new float[block_len];

  // 最坏情况下，只需 nprocs 轮即可完成排序
  for (int i = 0; i < nprocs; i++) {

    // 第一阶段：偶奇排序
    MPI_Request req[2];

    if (rank % 2) {
      // 奇数进程
      // 首先发送 1 个数据作比较
      MPI_Irecv(buffer, 1, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD, &req[0]);
      MPI_Isend(data, 1, MPI_FLOAT, rank - 1, 1, MPI_COMM_WORLD, &req[1]);
      MPI_Waitall(2, req, nullptr);

      // 若数据不是有序的，则交换余下数据
      if (data[0] < buffer[0]) {
        MPI_Irecv(buffer, block_len_left, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD, &req[0]);
        MPI_Isend(data, block_len, MPI_FLOAT, rank - 1, 1, MPI_COMM_WORLD, &req[1]);
        MPI_Waitall(2, req, nullptr);
```

```
        // 奇数进程保留大的
        get_large(data, block_len, buffer, block_len_left, tmp);
      }

    } else if (rank + 1 < nprocs) {
      // 偶数进程
      MPI_Irecv(buffer, 1, MPI_FLOAT, rank + 1, 1, MPI_COMM_WORLD,
&req[0]);
      MPI_Isend(data + block_len - 1, 1, MPI_FLOAT, rank + 1, 0,
MPI_COMM_WORLD, &req[1]);
      MPI_Waitall(2, req, nullptr);

      // 若数据不是有序的，则交换余下数据
      if (data[block_len - 1] > buffer[0]) {
        MPI_Irecv(buffer, block_len_right, MPI_FLOAT, rank + 1, 1,
MPI_COMM_WORLD, &req[0]);
        MPI_Isend(data, block_len, MPI_FLOAT, rank + 1, 0,
MPI_COMM_WORLD, &req[1]);
        MPI_Waitall(2, req, nullptr);
        // 偶数进程保留小的
        get_small(data, block_len, buffer, block_len_right, tmp);
      }

    }

    // 第二阶段：奇偶排序
    if (rank % 2) {
      if (!last_rank) {
        // 奇数进程，同样先发送一个数据作比较
        MPI_Irecv(buffer, 1, MPI_FLOAT, rank + 1, 1, MPI_COMM_WORLD,
&req[0]);
        MPI_Isend(data + block_len - 1, 1, MPI_FLOAT, rank + 1, 0,
MPI_COMM_WORLD, &req[1]);
        MPI_Waitall(2, req, nullptr);

        // 若数据不是有序的，则交换余下数据
        if (data[block_len - 1] > buffer[0]) {
          MPI_Irecv(buffer + 1, block_len_right - 1, MPI_FLOAT, rank
+ 1, 1, MPI_COMM_WORLD, &req[0]);
          MPI_Isend(data, block_len - 1, MPI_FLOAT, rank + 1, 0,
MPI_COMM_WORLD, &req[1]);
          MPI_Waitall(2, req, nullptr);
          // 奇数进程保留小的
          get_small(data, block_len, buffer, block_len_right, tmp);
        }
```

```
110         }
111       } else if (rank > 0) {
112         // 偶数进程
113         MPI_Irecv(buffer + block_len_left - 1, 1, MPI_FLOAT, rank - 1,
    0, MPI_COMM_WORLD, &req[0]);
114         MPI_Isend(data, 1, MPI_FLOAT, rank - 1, 1, MPI_COMM_WORLD,
    &req[1]);
115         MPI_Waitall(2, req, nullptr);
116 
117         // 若数据不是有序的，则交换余下数据
118         if (data[0] < buffer[block_len_left - 1]) {
119           MPI_Irecv(buffer, block_len_left - 1, MPI_FLOAT, rank - 1, 0,
    MPI_COMM_WORLD, &req[0]);
120           MPI_Isend(data + 1, block_len - 1, MPI_FLOAT, rank - 1, 1,
    MPI_COMM_WORLD, &req[1]);
121           MPI_Waitall(2, req, nullptr);
122           // 偶数进程保留大的
123           get_large(data, block_len, buffer, block_len_left, tmp);
124         }
125       }
126 
127     }
128 
129     delete[] buffer;
130     delete[] tmp;
131 }
```

- 整体而言无额外优化。

性能

- 在元素数量 $n = 100000000$ 的情况下不同进程数运行 `sort` 函数的时间如下：

机器数 N	进程数 P	运行时间/ms	加速比
1	1	12496.814	1.000
1	2	6625.917	1.886
1	4	3534.059	3.536
1	8	1982.254	6.304
1	16	1220.961	10.235
2	16	836.900	14.932