PA1 Report

宋建昊 2022010853

代码

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

// 与前一个进程比较并排序
inline int sort_prev(float*& data, float*& recv_buf, float*& return_buf, int
block_len, int prev_len, int rank) {
    if(prev_len == 0) return 0;  // 如果没有前一块数据，直接返回

    float prev_max;
    // 与前一个进程交换最大值，检查是否需要排序
    MPI_Sendrecv(&data[0], 1, MPI_FLOAT, rank - 1, 0,
                 &prev_max, 1, MPI_FLOAT, rank - 1, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    if(prev_max <= data[0]) return 0;  // 如果已经有序，无需继续

    // 非阻塞发送当前块到前一个进程
    MPI_Request req;
    MPI_Isend(data, block_len, MPI_FLOAT, rank - 1, 1, MPI_COMM_WORLD, &req);
    // 接收前一个进程的块
    MPI_Recv(recv_buf, prev_len, MPI_FLOAT, rank - 1, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

    // 使用指针优化合并排序，并在 MPI_Wait 前开始计算以重叠通信
    float* ptr_data = data + block_len - 1;
    float* ptr_recv = recv_buf + prev_len - 1;
    float* ptr_return = return_buf + block_len - 1;
    while(ptr_return >= return_buf) {
        if(ptr_data >= data && (ptr_recv < recv_buf || *ptr_data > *ptr_recv)) {
            *ptr_return-- = *ptr_data--;
        } else {
            *ptr_return-- = *ptr_recv--;
        }
    }
    MPI_Wait(&req, MPI_STATUS_IGNORE);  // 确保发送完成，此时计算已完成一部分或全部
    return 1;
}

// 与后一个进程比较并排序
inline int sort_next(float*& data, float*& recv_buf, float*& return_buf, int
block_len, int next_len, int rank) {
    if(next_len == 0) return 0;  // 如果没有后一块数据，直接返回

    float next_min;
```

```
    // 与后一个进程交换最小值，检查是否需要排序
    MPI_Sendrecv(&data[block_len - 1], 1, MPI_FLOAT, rank + 1, 0,
                 &next_min, 1, MPI_FLOAT, rank + 1, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    if(data[block_len - 1] <= next_min) return 0;  // 如果已经有序，无需继续

    // 非阻塞发送当前块到后一个进程
    MPI_Request req;
    MPI_Isend(data, block_len, MPI_FLOAT, rank + 1, 1, MPI_COMM_WORLD, &req);
    // 接收后一个进程的块
    MPI_Recv(recv_buf, next_len, MPI_FLOAT, rank + 1, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

    // 使用指针优化合并排序，并在 MPI_Wait 前开始计算以重叠通信
    float* ptr_data = data;
    float* ptr_recv = recv_buf;
    float* ptr_return = return_buf;
    float* end_return = return_buf + block_len;
    while(ptr_return < end_return) {
        if(ptr_recv >= recv_buf + next_len) {
            while(ptr_return < end_return) {
                *ptr_return++ = *ptr_data++;
            }
        } else {
            *ptr_return++ = (*ptr_data < *ptr_recv) ? *ptr_data++ : *ptr_recv++;
        }
    }
    MPI_Wait(&req, MPI_STATUS_IGNORE);  // 确保发送完成，此时计算已完成一部分或全部
    return 1;
}

// 奇偶排序实现（保持不变，仅展示上下文）
void Worker::sort() {
    // 对本地块进行初始排序
    std::sort(data, data + block_len);
    if(nprocs == 1) return;  // 只有一个进程，无需并行排序
    if(out_of_range) return; // 当前进程没有数据需要排序

    const int block_size = ceiling(n, nprocs);  // 计算块大小
    const int worker_num = ceiling(n, block_size);  // 计算活跃工作进程数
    const int prev_size = rank > 0 ? block_size : 0;  // 前一块的大小
    const int next_size = (rank >= worker_num - 1) ? 0 :
                          (rank == worker_num - 2 ? n - (worker_num - 1) *
block_size : block_size);
    bool is_even = !(rank & 1);  // 判断当前进程是奇数还是偶数

    if(block_len == 0) return;
    // 为合并操作分配临时缓冲区
    float* return_buf = new float[block_size * 2];
    float* recv_buf = return_buf + block_size;

    // 奇偶排序阶段
    for(int i = 0; i < worker_num; i += 2) {
        if(is_even) {
            if(sort_next(data, recv_buf, return_buf, block_len, next_size, rank))
```

```
                std::swap(data, return_buf);
            if(sort_prev(data, recv_buf, return_buf, block_len, prev_size, rank))
                std::swap(data, return_buf);
        } else {
            if(sort_prev(data, recv_buf, return_buf, block_len, prev_size, rank))
                std::swap(data, return_buf);
            if(sort_next(data, recv_buf, return_buf, block_len, next_size, rank))
                std::swap(data, return_buf);
        }
    }

    delete[] return_buf;  // 释放内存
}
```

优化策略

通信与计算重叠

- **优化方式**：在 sort_prev 和 sort_next 函数中，使用了非阻塞发送 MPI_Isend 来发送数据，并在等待通信完成（MPI_Wait）之前开始执行合并排序的计算。这种方式允许计算和通信并行进行，从而减少了等待时间。
- **实现细节**：
  - 在发送数据后，立即开始使用指针操作进行合并排序，而不是等待通信完成。例如，在 sort_prev 中：

    ```
    MPI_Isend(data, block_len, MPI_FLOAT, rank - 1, 1, MPI_COMM_WORLD,
    &req);
    MPI_Recv(recv_buf, prev_len, MPI_FLOAT, rank - 1, 1, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
    // 在 MPI_Wait 之前开始合并排序
    float* ptr_data = data + block_len - 1;
    float* ptr_recv = recv_buf + prev_len - 1;
    float* ptr_return = return_buf + block_len - 1;
    while(ptr_return >= return_buf) {
        // 合并逻辑
    }
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    ```

  - 这种方法利用了 CPU 在等待通信完成时的空闲周期，特别是在网络延迟较高的情况下效果显著。
- **性能提升**：通过重叠通信和计算，减少了进程间的同步开销，尤其在数据块较大或网络带宽有限时更为明显。

指针操作优化合并排序

- **优化方式**：在合并排序过程中，使用了指针操作（ptr_data、ptr_recv、ptr_return）而不是数组索引，避免了额外的索引计算和边界检查。
- **实现细节**：
  - 在 sort_prev 中，从高地址向低地址合并：

```
while(ptr_return >= return_buf) {
    if(ptr_data >= data && (ptr_recv < recv_buf || *ptr_data >
*ptr_recv)) {
        *ptr_return-- = *ptr_data--;
    } else {
        *ptr_return-- = *ptr_recv--;
    }
}
```

- 在 sort_next 中，从低地址向高地址合并：

```
while(ptr_return < end_return) {
    if(ptr_recv >= recv_buf + next_len) {
        while(ptr_return < end_return) {
            *ptr_return++ = *ptr_data++;
        }
    } else {
        *ptr_return++ = (*ptr_data < *ptr_recv) ? *ptr_data++ :
*ptr_recv++;
    }
}
```

- **性能提升**：指针操作减少了 CPU 的计算开销（如数组索引的加法和乘法），提高了内存访问效率，尤其在处理大数据块时能显著降低循环内的指令数。

特殊情况特判

- **优化方式**：在 sort_prev 和 sort_next 中，通过比较相邻块的最大值和最小值来判断是否需要进行完整的排序和合并操作。如果数据已经有序，则直接返回，避免不必要的计算和通信。
- **实现细节**：
  - 在sort主函数中：

```
    if(nprocs == 1) return;  // 只有一个进程，无需并行排序
    if(out_of_range) return; // 当前进程没有数据需要排序
    if(block_len == 0) return; //当前块没有被分配待排序数组，无需进行操作
```

  - 在 sort_prev 中：

```
MPI_Sendrecv(&data[0], 1, MPI_FLOAT, rank - 1, 0,
             &prev_max, 1, MPI_FLOAT, rank - 1, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
if(prev_max <= data[0]) return 0;  // 如果前块最大值小于当前块最小值，无需排序
```

  - 在 sort_next 中：

```
MPI_Sendrecv(&data[block_len - 1], 1, MPI_FLOAT, rank + 1, 0,
             &next_min, 1, MPI_FLOAT, rank + 1, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
if(data[block_len - 1] <= next_min) return 0;  // 如果当前块最大值小于后块最小值，无需排序
```

- **性能提升**：这种检查机制避免了不必要的通信和合并操作，尤其在数据部分有序的情况下，可以显著减少计算量和通信开销。

性能测试

$N \times P$ 表示 N 台机器，每台机器 P 个进程

排序问题规模 = 100000000

N / P	运行时间(ms)	相对单进程的加速比
1/1	12257.231000	1.0000
1/2	6476.153000	1.8927
1/4	3392.631000	3.6129
1/8	1904.723000	6.4352
1/16	1133.656000	10.8121
2/16	1137.781000	10.7729