Parallel Programming

Homework 1: Odd-Even Sort

111065524 李懷

Implementation:

Allocate elements for each rank:

```
138     int i, rank, size, namelen;
139     char name[MPI_MAX_PROCESSOR_NAME];
140     MPI_Status stat;
141
142     MPI_Init(&argc, &argv);
143     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
144     MPI_Comm_size(MPI_COMM_WORLD, &size);
145     // MPI_Get_processor_name(name, &namelen);
146
147     double tik, tok;
148
149     tik = MPI_Wtime();
150
151     MPI_File input_file, output_file;
152
153     long long chunk = n / size;     // base chunk size
154     int r = (int)(n % size);        // leftover: how many ranks get +1
155     // printf("n = %zu, size = %d, chunk = %lld, r = %d\n", n, size, chunk, r);
156     if (rank < r) {
157         chunk += 1;                 // first r ranks get +1
158     }
159
160     printf("rank = %d\n chunk = %lld\n", rank, chunk);
161     int last_rank = size-1;
162     if (n <= size) {
163         last_rank = n-1;
164     }
165
166     vector<float> data(chunk);
```

chunk: how many elements each rank

Total of n elements, we allocate each rank with n/size(# of processes)

We have r leftover numbers unallocated, so first r ranks get +1 for their chunk, r = n%size, so if n < # of processes, the n ranks get 1 chunk, rankN and above get no element.

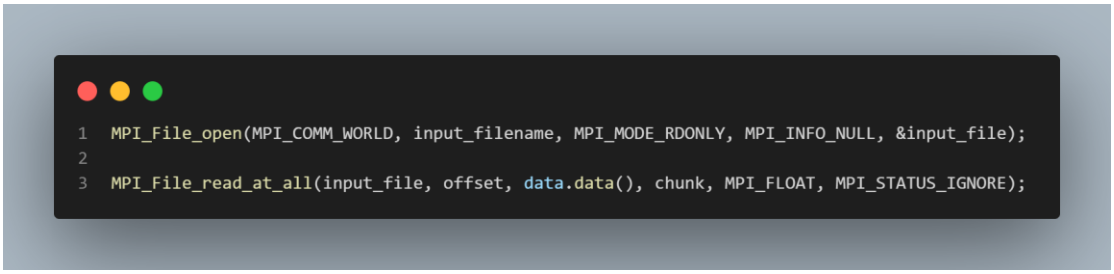
Example: 100 of total elements to sort，given 3 processes，rank0 will get 34，rank1~2 will get 33.

Set the MPI offset:

```
165    MPI_Offset offset = (MPI_Offset)rank * (MPI_Offset)chunk * (MPI_Offset)sizeof(float); // byte offset, rank0: 0~chunk*4bytes
166    if (rank >= r) {
167        offset += (MPI_Offset)r * (MPI_Offset)sizeof(float); // first r ranks has +1 chunk
168    }
```

Set each rank mpi offset to read the correct parts of the binary file for each rank

Read the file:



```
1  MPI_File_open(MPI_COMM_WORLD, input_filename, MPI_MODE_RDONLY, MPI_INFO_NULL, &input_file);
2
3  MPI_File_read_at_all(input_file, offset, data.data(), chunk, MPI_FLOAT, MPI_STATUS_IGNORE);
```

Sorting:

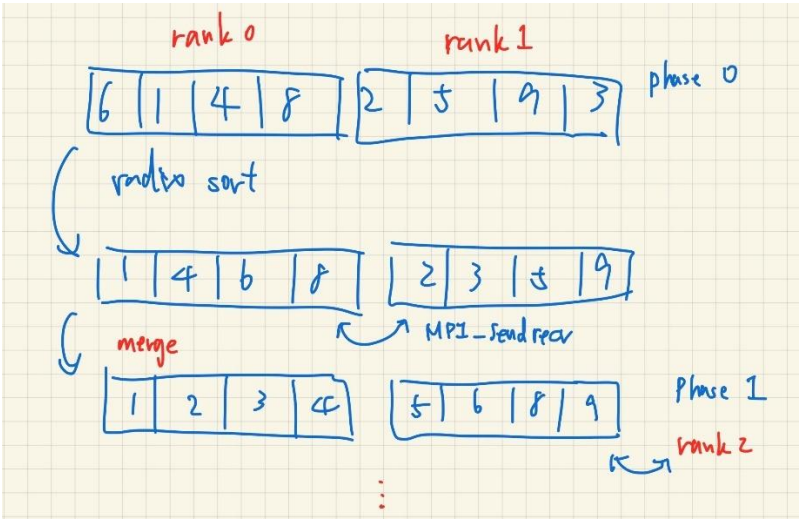
```
190    while (global_swapped) {
191        bool local_swapped = false;
192        // sort each rank array locally after swaps
193
194        if (chunk > 1 && phase == 0){
195            radix_sort_floats(data.data(), chunk); // n
196        }
```

Each rank will go in this while loop, at phase 0, each rank will sort its local data chunk first, we implement radix sort for the local sort first (see appendix), which is a non-comparative sorting algorithm, complexity is O(d×(n+k)), where d is 4(4 bytes for float), k is 256(2^8, 8bits per byte), which is generally faster than O(nlogn) sort

Odd even sort:

```cpp
if (chunk>0) {
            int peerChunk = chunk;
            // Even phase: (0,1), (2,3), ...; Odd phase: (1,2), (3,4), ...
            if (rank < last_rank &&
                ((phase % 2 != 0 && rank % 2 != 0) ||
                (phase % 2 == 0 && rank % 2 == 0))) { //current rank is the former of the pair
                    if (rank < r && rank + 1 >= r){
                        peerChunk -= 1; // next rank has 1 less chunk
                        recvBuf.resize(peerChunk);
                    }
                    // printf("rank %d sending %d floats to rank %d which has %d\n", rank, chunk, rank + 1, peerChunk);

                    MPI_Sendrecv(data.data(), chunk, MPI_FLOAT, rank + 1, 0,
                            recvBuf.data(), peerChunk, MPI_FLOAT, rank + 1, 0,
                            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    // MPI_Sendrecv(&data[chunk - 1],   1, MPI_FLOAT, rank + 1, 0,
                    //             &recvBuf[0],1, MPI_FLOAT, rank + 1, 0,
                    //             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

                    local_swapped = (data[chunk - 1] > recvBuf[0]);

                    // printf("rank %d local_swapped = %d\n", rank, local_swapped);
                    if (local_swapped) {

                        // // std::merge(recvBuf.begin(), recvBuf.end(), data.begin(), data.end(), tmp.begin()); // merge sort

                        // // // keep first half (smaller elements), replace rank's data
                        // // std::copy(tmp.begin(), tmp.begin() + chunk, data.begin());

                        int i = 0, j = 0, k = 0;
                        while (k < chunk) {
                            if (j >= peerChunk || (i < chunk && data[i] <= recvBuf[j])){
                                tmp[k++] = data[i++];
                            } else {
                                tmp[k++] = recvBuf[j++];
                            }
                        }
                        std::memcpy(data.data(), tmp.data(), size_t(chunk) * sizeof(float)); // copy from tmp to current rank data

                    }
            } else if (rank > 0 &&
                ((phase % 2 != 0 && rank % 2 == 0) ||
                (phase % 2 == 0 && rank % 2 != 0))) { // current rank is the latter of the pair
                    if (rank >= r && rank - 1 < r){
                        peerChunk += 1; // previous rank has 1 more chunk
                        recvBuf.resize(peerChunk);
                    }
                    // printf("rank %d sending %d floats to rank %d which has %d\n", rank, chunk, rank - 1, peerChunk);

                    // tmp.resize(chunk + peerChunk);
                    // tmp.resize(chunk);

                    MPI_Sendrecv(data.data(), chunk, MPI_FLOAT, rank - 1, 0,
                            recvBuf.data(), peerChunk, MPI_FLOAT, rank -1, 0,
                            MPI_COMM_WORLD, MPI_STATUS_IGNORE);

                    // MPI_Sendrecv(&data[0], 1, MPI_FLOAT, rank - 1, 0,
                    //             &recvBuf[peerChunk-1],1, MPI_FLOAT, rank - 1, 0,
                    //             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

                    local_swapped = (data[0] < recvBuf[peerChunk-1]);
                    // printf("rank %d local_swapped = %d\n", rank, local_swapped);
                    if (local_swapped) {
                        // // std::merge(data.begin(), data.end(), recvBuf.begin(), recvBuf.end(), tmp.begin()); // merge sort

                        // std::copy(tmp.end() - chunk, tmp.end(), data.begin());
                        int i = chunk - 1, j = peerChunk - 1, k = chunk - 1;
                        while (k >= 0) {
                            if (j < 0 || (i >= 0 && data[i] >= recvBuf[j])) {
                                tmp[k--] = data[i--];
                            }else{
                                tmp[k--] = recvBuf[j--];
                            }
                        }
                        std::memcpy(data.data(), tmp.data(), size_t(chunk) * sizeof(float));

                    }
            }
```

Rough illustration:



At even phase, e.g. when phase = 0,2,4,…, we exchange datas between rank pair (0,1), (2,3), (4,5),…, at even phase, we exchange between (1,2), (3,4), (5,6),

So for even phase, even rank will send their data to its right neighbor, like rank0 send to rank1 etc, odd rank will send to its left neighbor, for the last rank, it has no right neighbor, so no need to send; for odd phase, odd rank send to right, even rank send to left, for rank0, no need to send, we adjust recvBuf size if exchange neighbor's chunk size is different.

After MPI_Sendrecv for each pair, each rank pair compare their edge element, e.g. for rank pair (0,1), compare rank0's last element and rank1's first element, if data[chunk - 1] > recvBuf[0], we merge sort two vectors, since two vectors are already sorted at phase 0, so complexity for this step is O(n), then copy the first chunk elements to rank0's data, now all elements of rank0 is sorted and smaller than all elements of rank1, this will apply to all rank pair at each phase.

```
if (phase > 0){
            // if no swaps happened for all ranks, stop
            int local = local_swapped ? 1 : 0;
            int any = 0;
            MPI_Allreduce(&local, &any, 1, MPI_INT, MPI_LOR, MPI_COMM_WORLD);
            global_swapped = (any != 0);
        }
```

After each rank finishes talking with its neighbor, before going to next phase, we check at each phase(except phase0) if there is any data swapping happens for all rank pair, if no pair has exchanged for that phase, means entire array sorted, we exit the while loop (since previous phase(e.g. odd phase) has done swapping, and current phase(even phase) has no swapping, means the array is all sorted and there is no need to check next phase).

Finally, we write all ranks' data back to one file using MPI_File_write_at_all:

```
MPI_File_open(MPI_COMM_WORLD, output_filename,
            MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &output_file);

MPI_File_write_at_all(output_file, offset, data.data(), (int)chunk, MPI_FLOAT, MPI_STATUS_IGNORE);
// printf("rank %d wrote %d floats, write from %d\n", rank, chunk, sizeof(float) * (int)chunk * rank);
// --- close file ---
MPI_File_close(&output_file);

MPI_Finalize();
return 0;
```

Experiment & Analysis:

i. Methodology

System Specs:

We run our experiments on the course cluster.

Performance Metrics:

For each job, we use nsys commands(provided by TA) to generate .nsys-rep and csv for each rank.

We then read each csv file and record the longest elapsed time of all rank, this will be the total duration time for the job.

We then find the average communication time and io time of all ranks,

$$comm_{\_all} = \frac{1}{\#ranks} * \sum_{r} comm_{\_r}$$

$$io_{\_all} = \frac{1}{\#ranks} * \sum_{r} io_{\_r}$$

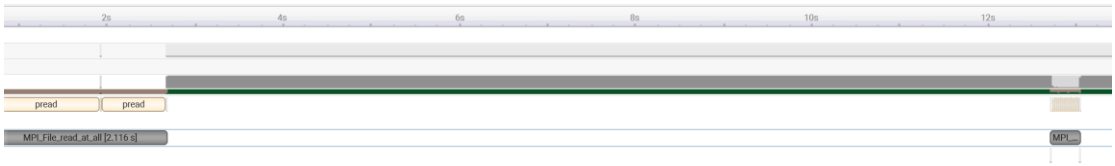
$comm_{\_\mathrm{r}}$: includes: MPI_Sendrecv and MPI_Allreduce

$io_{\_\mathrm{r}}$: includes: MPI_File_open, MPI_File_read_at_all and MPI_File_write_at_all etc

For cpu time, it's total time minus communication and io time:

$$cpu_{\_all} = Total - comm_{\_all} - io_{\_all}$$

Since CPU time will be the blank duration without any MPI event, like:



ii. Plots: Speedup Factor & Profile

- **Experimental Method:**

We use 2 testcases, one is 33.in file, which contains 536869888 floats, provided by Tas, and we also generate a binary file of 4GB, containing 1073741824 of floats, however, this one is in descending order, which is worst case for odd-even sort, so it should really test out the MPI communication time to better see the difference between different # of processes, since to be able to see communication time bottleneck, we need high process count, but we cannot set processes(and core count) too high because we can only run maximum of 36 cores(3 nodes?) with srun -N -n -c, so 48 is as high as we can set.

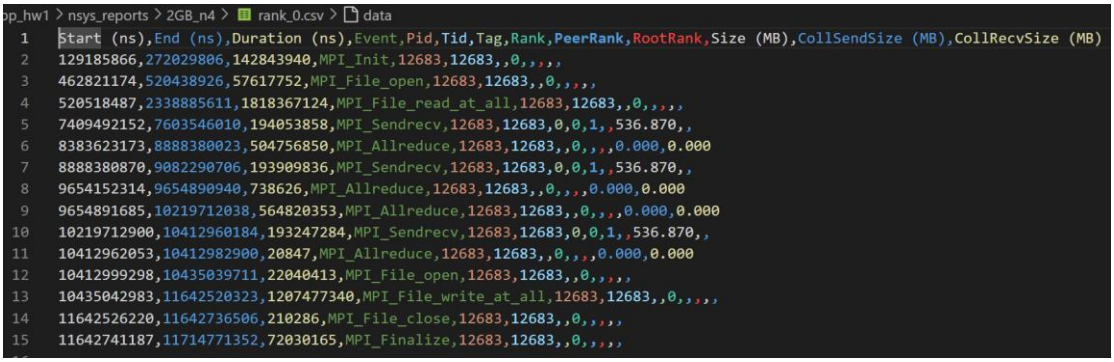
We run it with 1, 2, 4, 12, 24, 48 of processes, with 1 core, and let slurm allocate the nodes, according to the lab, each node has 6 cores 12 threads, so job run with 1 core, 24 and 48 processes should use 2 and 4 nodes.

- **Performance Measurement:**

We use Nsight to profile our mpi applications(with script provided by TAs)
Then export MPI event trace to csv with:

```
nsys stats -r mpi_event_trace --format csv
```

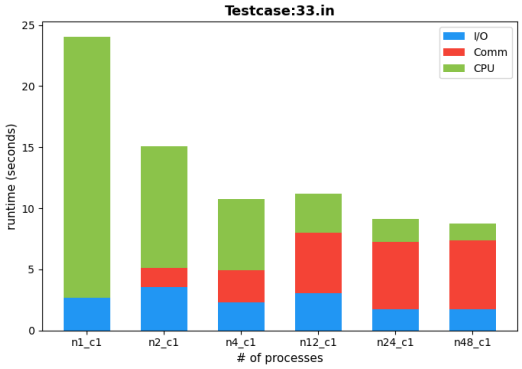
MPI event trace csv table of one rank, this shows the duration of each mpi event:

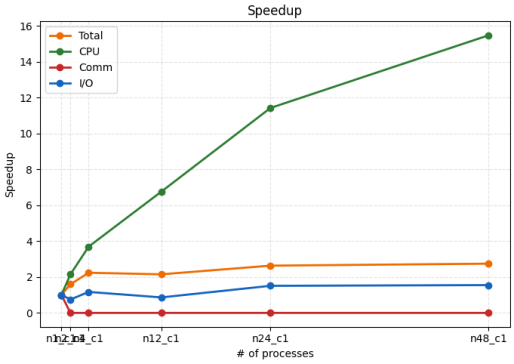


We then use pandas to read csv and compute execution time, communication time and IO time, then use matplotlib to visualize.

- **Analysis of Results:**

1. Graph for 1st testcase(33.in):

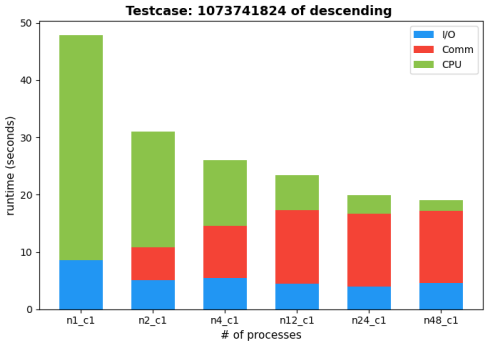


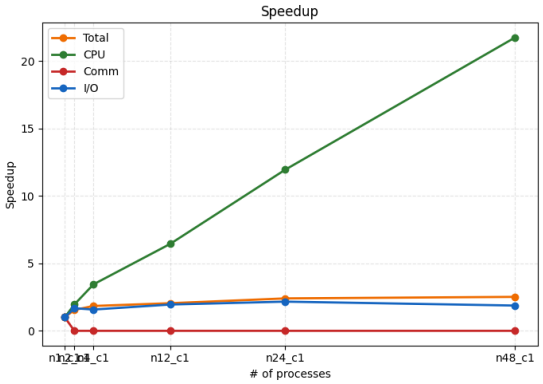


n : # of process, c : # of cores.

2. Graph for 2nd testcase(4GB of descending floats):

We generate a file containing 1073741824 of floats in descending order, this will better test out the MPI communication bottleneck of our program without using a super large-sized file, since it will require a lot of transposition between ranks.





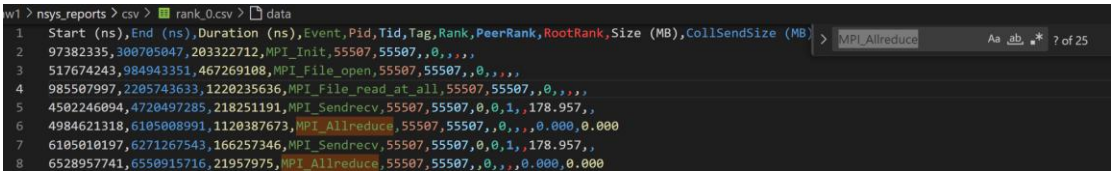
As we can see comm time is significantly higher for higher process count, as the number of exchange times between ranks is higher.

With the speedup figure, we can check how well the program scale with process count:

As we can see, despite the CPU time speedup scales almost linearly with process count, but the Comm time cannot get any lower and even increases, which becomes bottleneck at higher process count.

This is especially obvious for 2nd testcase as it is stressful for odd-even sort, we can see there is all lot of MPI_Sendrecv and Allreduce events:

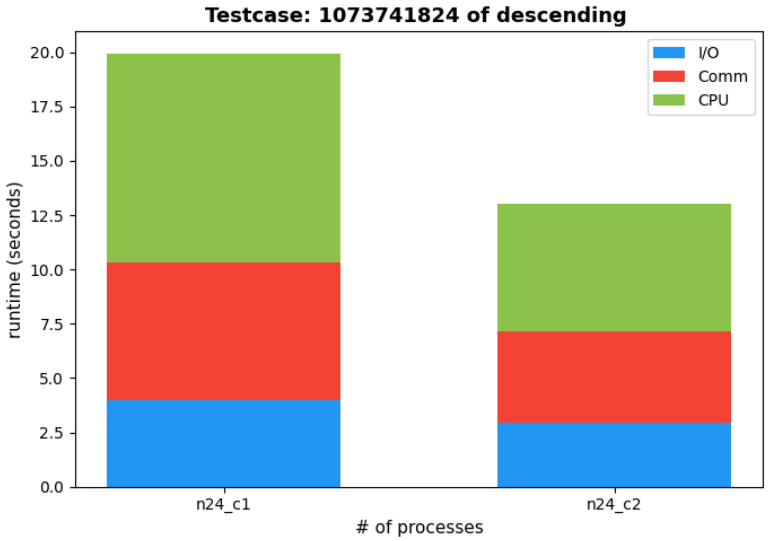
n=24



n=48

However, despite the count of MPI exchange events increase linearly in this case, the comm time does not increase linearly(despite still being a bottleneck), it is because, as process count increases, the exchange data chunk size decrease, so it kind of balances out.

Also we check how well the program scale with multiple cores with the same process count:



More cores can reduce CPU time, which is mainly from the initial sort, but I can also affect communication time, despite the number of MPI exchange event is the same, the exchange time can be improved with core count.

- Optimization Strategies:
  - Based on the analysis results, propose potential optimization strategies.

    Since the CPU time scales well, and we already use radix sort, and for IO time, i.e. MPI read and write file operations, it does not increase with process count(however, we can still improve it by setting the MPI_Info), so optimization should focus on MPI communication time as it becomes a bottleneck as process count increases.

    So the potential strategies are:

    1. instead of merging data chunks between ranks when there is need to swap, we can just compare where there is overlap, so right now we are comparing left rank's last element and right rank's first element to see if there is need for swapping, then perform merge that is linear time, however, we can do binary search with rank's last element(or first element depending on left or right) as target, then find the right place to start merging, this should reduce the merging time as the chunk size becomes very large.

    2. We can also try nonblocking MPI operations, mpi_sendrecv is blocking, where it performs a send and a receive simultaneously, but the call does not return until both operations have completed (i.e., the message has been sent and the corresponding receive has finished). So we can try MPI_Isend and MPI_Irecv.

iii. Discussion (Must base on the results in your plots)

1. Compare I/O, CPU, Network performance. Which is/are the bottleneck(s)? Why? How could it be improved?

Comm time will become bottleneck for both cases , as we can see, as the process count increases, the comm time also increase, despite the CPU time scales linearly better with process count, the total time cannot get any significantly lower after certain process count.

*Note that to test out bottleneck on communication time, we need higher processes count to show more obvious results, but we cannot run total CPU count more than 48, because we cannot run on more than 4 nodes, so we set max process count to 48 and 1 core .

2. Compare scalability. Does your program scale well? Why or why not? How can you achieve better scalability? You may discuss the two implementations separately or together.

From the results, the program shows good computational scalability but limited overall scalability. As the number of processes increases from 1 to 48, the total runtime decreases initially but then flattens, while the proportion of communication time grows significantly. The CPU component scales almost linearly, indicating that the computation is well parallelized, but the total speedup plateaus around 3× because communication overhead and synchronization between ranks dominate at higher process counts. In other words, the program becomes communication-bound beyond about 12–24 processes. To achieve better scalability, the implementation should reduce communication frequency, overlap computation with communication using nonblocking MPI operations, and ensure balanced workloads or larger per-process work to maintain a high compute-to-communication ratio.

Experiences / Conclusion

Conclusion:

We use Nsight to profile our MPI program and visualize the datas so we can better understand the detailed performance of our code, the results showed that the algorithm scales reasonably well, but communication overhead becomes a limiting factor for program to scale even better as the number of processes increases. While the workload is evenly distributed, synchronization between processes and collective operations (like MPI_Allreduce) significantly affect overall runtime and has room for optimization.

Overall, this assignment strengthened my understanding of how parallel sorting behaves differently from sequential algorithms, not only algorithmically but also in terms of hardware and communication.

Things I've learned:

- How to decompose a sorting problem and distribute data among MPI ranks.
- How each MPI function (e.g., MPI_Sendrecv, MPI_Allreduce, and MPI_File_*) works.
- The importance of communication–computation overlap and minimizing synchronization in parallel algorithms.
- How to use Nsight Systems to profile a MPI program and interpret the results (CPU vs communication vs I/O time), without using MPI_Wtime to wrap every block of code.
- How hardware configuration influences communication cost.

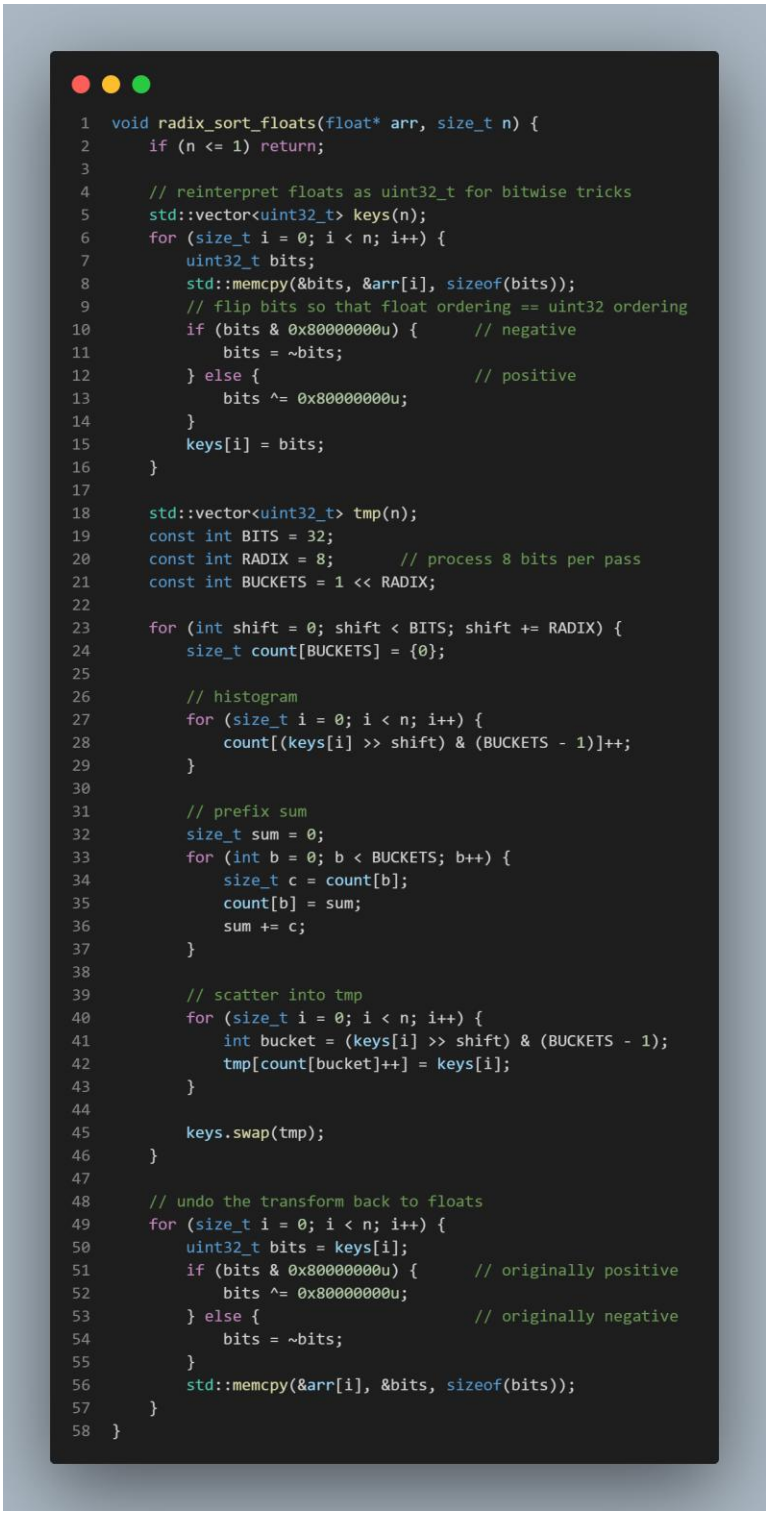
Difficulties:

- Getting MPI communication patterns correct — especially ensuring correct neighbor exchange between even and odd phases.
- Handling termination detection (using MPI_Allreduce to check if any rank performed a swap).
- Managing file I/O with MPI_File_ functions*, it took me some time to finally read the file correctly because it's my first time with mpi.
- Using Nsight Systems to analyze the data, the GUI is a bit daunting
- Trying to increase performance

- Performance tuning — since performance bottleneck can be from CPU/communication/IO time, it takes time to identify which is bottlenecking, so I have to check each one to the correct part to improve.

Appendix:

Radix sort implementation:



```cpp
void radix_sort_floats(float* arr, size_t n) {
    if (n <= 1) return;

    // reinterpret floats as uint32_t for bitwise tricks
    std::vector<uint32_t> keys(n);
    for (size_t i = 0; i < n; i++) {
        uint32_t bits;
        std::memcpy(&bits, &arr[i], sizeof(bits));
        // flip bits so that float ordering == uint32 ordering
        if (bits & 0x80000000u) {      // negative
            bits = ~bits;
        } else {                       // positive
            bits ^= 0x80000000u;
        }
        keys[i] = bits;
    }

    std::vector<uint32_t> tmp(n);
    const int BITS = 32;
    const int RADIX = 8;       // process 8 bits per pass
    const int BUCKETS = 1 << RADIX;

    for (int shift = 0; shift < BITS; shift += RADIX) {
        size_t count[BUCKETS] = {0};

        // histogram
        for (size_t i = 0; i < n; i++) {
            count[(keys[i] >> shift) & (BUCKETS - 1)]++;
        }

        // prefix sum
        size_t sum = 0;
        for (int b = 0; b < BUCKETS; b++) {
            size_t c = count[b];
            count[b] = sum;
            sum += c;
        }

        // scatter into tmp
        for (size_t i = 0; i < n; i++) {
            int bucket = (keys[i] >> shift) & (BUCKETS - 1);
            tmp[count[bucket]++] = keys[i];
        }

        keys.swap(tmp);
    }

    // undo the transform back to floats
    for (size_t i = 0; i < n; i++) {
        uint32_t bits = keys[i];
        if (bits & 0x80000000u) {      // originally positive
            bits ^= 0x80000000u;
        } else {                       // originally negative
            bits = ~bits;
        }
        std::memcpy(&arr[i], &bits, sizeof(bits));
    }
}
```