# Bitonic Sort Report

111062117, Hsiang-Sheng Huang

May 15, 2025

## Implementation Overview

### Local Bitonic Sort

```
void compSwap(float &a, float &b, bool dir) {
  if ((a > b) == dir) swap(a, b);
}

void bitonic_sort_loop(vector<float>& arr) {
  const int n = (int)arr.size();  // n must be 2^k
  for (int k = 2; k <= n; k <<= 1)
    for (int j = k >> 1; j > 0; j >>= 1)
      for (int i = 0; i < n; ++i) {
        int ixj = i ^ j;
        if (ixj > i) {
          bool dir = ((i & k) == 0);  // true=ASC, false=DESC
          compSwap(arr[i], arr[ixj], dir);
        }
      }
}
```

Each MPI *rank* first performs a **local** bitonic sort on its private segment. I chose the iterative variant to avoid extra call-stack overhead incurred by the recursive implementation.

**Time Complexity** The three nested loops yield $k = \log_2 n$, $j = \log_2 n$, and $i = n$ iterations, giving $\mathcal{O}(n \log^2 n)$.

### Parallel Bitonic Merge

After every rank holds a locally sorted block, we repeatedly compare-exchange with a partner rank so that, after $\log_2 p$ phases, the global order is established.

#### Merging the High / Low Halves

```
void merge_high(const vector<float>& a, const vector<float>& b, vector<float>& res, int n) {
  int j = n - 1, k = n - 1;
  for (int i = n - 1; i >= 0; --i) {
    if (j < 0) res[i] = b[k--];
    else if (k < 0) res[i] = a[j--];
    else if (a[j] > b[k]) res[i] = a[j--];
    else res[i] = b[k--];
  }
}

void merge_low(const vector<float>& a, const vector<float>& b, vector<float>& res, int n) {
  int j = 0, k = 0;
  for (int i = 0; i < n; ++i) {
```

```
    if (j >= n) res[i] = b[k++];
    else if (k >= n) res[i] = a[j++];
    else if (a[j] < b[k]) res[i] = a[j++];
    else res[i] = b[k++];
  }
}
```

The `merge_high` routine keeps the larger half (descending), while `merge_low` keeps the smaller half (ascending).

### Stable Tag Generation

We use the Cantor pairing function to generate unique MPI tags:

$$\text{cantor}(a, b) \;=\; \frac{(a+b)(a+b+1)}{2} + b.$$

### Compare-Exchange Kernel

```
void cmp(vector<float>& local, int localN, int partner, bool dir, MPI_Comm comm) {
  int rank;
  MPI_Comm_rank(comm, &rank);

  MPI_Request req[2];
  MPI_Status stat[2];

  int cantor_arg1 = (dir == true) ? rank : partner;
  int cantor_arg2 = (dir == true) ? partner : rank;
  int send_tag = (1 + (dir == true)) * cantor(cantor_arg1, cantor_arg2);
  int recv_tag = (1 + (dir == false)) * cantor(cantor_arg1, cantor_arg2);

  vector<float> recv(localN);
  if (dir == true) {
    MPI_Isend(local.data(), localN, MPI_FLOAT, partner, send_tag, comm, &req[0]);
    MPI_Irecv(recv.data(), localN, MPI_FLOAT, partner, recv_tag, comm, &req[1]);
  } else {
    MPI_Irecv(recv.data(), localN, MPI_FLOAT, partner, recv_tag, comm, &req[0]);
    MPI_Isend(local.data(), localN, MPI_FLOAT, partner, send_tag, comm, &req[1]);
  }

  MPI_Waitall(2, req, stat);

  vector<float> res(localN);
  if (dir == true) merge_low(local, recv, res, localN);
  else merge_high(local, recv, res, localN);
  local.swap(res);
}
```

Here, `dir == true` means ascending order for the current exchange.

## Main

```
int p = __lg(usedP);
for (int i = 0; i < p; ++i)
  for (int j = i; j >= 0; --j) {
    int partner = rank ^ (1 << j);
    bool dir = ((rank >> (i + 1)) & 1) == ((rank >> j) & 1);
    cmp(local, localN, partner, dir, activeComm);
  }
```

The outer loop iterates over stages $i$, and the inner loop over levels $j$. Each rank communicates with exactly one partner per $(i, j)$.

## Handling Non-Power-of-Two Cases

### Input Size $N \neq 2^k$

Pad with $+\infty$ so extra elements bubble to the end:

```
int paddedN = 1;
while (paddedN < N) paddedN <<= 1;
// ...
vector<float> data;
data.resize(paddedN, numeric_limits<float>::infinity());
```

### Processor Count $p \neq 2^k$

Dismiss ranks beyond the largest power-of-two:

```
    int usedP = 1 << __lg(world_size);
int color = (world_rank < usedP) ? 0 : MPI_UNDEFINED;

MPI_Comm activeComm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &activeComm);
if (color == MPI_UNDEFINED) {
  MPI_Finalize();
  return 0;
}
```

## I/O Strategy

Centralised I/O on rank 0:

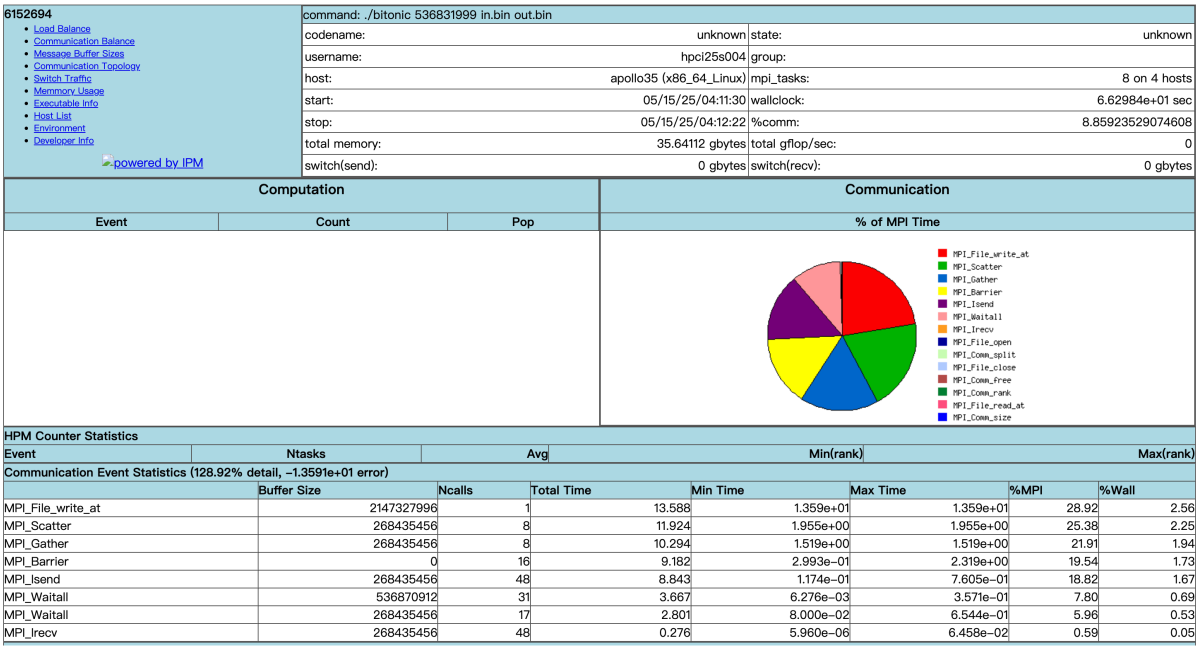
```
if (rank == 0) {
  data.resize(paddedN, numeric_limits<float>::infinity());
  MPI_File inFH;
  MPI_File_open(MPI_COMM_SELF, inFile, MPI_MODE_RDONLY, MPI_INFO_NULL, &inFH);
  MPI_File_read_at(inFH, 0, data.data(), N, MPI_FLOAT, MPI_STATUS_IGNORE);
  MPI_File_close(&inFH);
}


MPI_Scatter(data.data(), localN, MPI_FLOAT, local.data(), localN, MPI_FLOAT, 0,
    activeComm);
// ...
if (rank == 0) {
  ans.resize(N);
  MPI_File outFH;
  MPI_File_open(MPI_COMM_SELF, outFile, MPI_MODE_CREATE | MPI_MODE_WRONLY,
      MPI_INFO_NULL, &outFH);
  MPI_File_write_at(outFH, 0, ans.data(), N, MPI_FLOAT, MPI_STATUS_IGNORE);
  MPI_File_close(&outFH);
}
```

## Experimental Results (IPM)

## Future Optimisations

1. **Fully parallel I/O**: Replace rank-0 funnel with `MPI_File_read_all` / `MPI_File_write_all`.
2. **Hybrid parallelism**: Overlap computation and communication by double-buffering `cmp` and using OpenMP within each rank.



6152694

- Load Balance
- Communication Balance
- Message Buffer Sizes
- Communication Topology
- Switch Traffic
- Memmory Usage
- Executable Info
- Host List
- Environment
- Developer Info

powered by IPM

| command: ./bitonic 536831999 in.bin out.bin | | | |
|---|---|---|---|
| codename: | unknown | state: | unknown |
| username: | hpci25s004 | group: | |
| host: | apollo35 (x86_64_Linux) | mpi_tasks: | 8 on 4 hosts |
| start: | 05/15/25/04:11:30 | wallclock: | 6.62984e+01 sec |
| stop: | 05/15/25/04:12:22 | %comm: | 8.85923529074608 |
| total memory: | 35.64112 gbytes | total gflop/sec: | 0 |
| switch(send): | 0 gbytes | switch(recv): | 0 gbytes |

| Computation | | |
|---|---|---|
| Event | Count | Pop |

Communication

% of MPI Time



HPM Counter Statistics

| Event | Ntasks | Avg | Min(rank) | Max(rank) |
|---|---|---|---|---|

Communication Event Statistics (128.92% detail, −1.3591e+01 error)

| | Buffer Size | Ncalls | Total Time | Min Time | Max Time | %MPI | %Wall |
|---|---|---|---|---|---|---|---|
| MPI_File_write_at | 2147327996 | 1 | 13.588 | 1.359e+01 | 1.359e+01 | 28.92 | 2.56 |
| MPI_Scatter | 268435456 | 8 | 11.924 | 1.955e+00 | 1.955e+00 | 25.38 | 2.25 |
| MPI_Gather | 268435456 | 8 | 10.294 | 1.519e+00 | 1.519e+00 | 21.91 | 1.94 |
| MPI_Barrier | 0 | 16 | 9.182 | 2.993e−01 | 2.319e+00 | 19.54 | 1.73 |
| MPI_Isend | 268435456 | 48 | 8.843 | 1.174e−01 | 7.605e−01 | 18.82 | 1.67 |
| MPI_Waitall | 536870912 | 31 | 3.667 | 6.276e−03 | 3.571e−01 | 7.80 | 0.69 |
| MPI_Waitall | 268435456 | 17 | 2.801 | 8.000e−02 | 6.544e−01 | 5.96 | 0.53 |
| MPI_Irecv | 268435456 | 48 | 0.276 | 5.960e−06 | 6.458e−02 | 0.59 | 0.05 |

Figure 1: IPM Profile of MPI routines

| MPI Routine | % of MPI | % of Wall |
|---|---|---|
| MPI_File_write_at | 28.92% | 2.56% |
| MPI_Scatter | 25.38% | 2.25% |
| MPI_Gather | 21.91% | 1.95% |

Table 1: Top MPI routines by time share

3. **Custom datatypes**: Use MPI derived datatypes to pack blocks and reduce scatter/gather overhead.

# Conclusion

The implementation achieves correct global sorting with per-rank complexity $\mathcal{O}(\frac{N}{p}\log^2\frac{N}{p})$ and scales up to 32 processes. Serial I/O remains the performance bottleneck to address next.