PA1 Report

代码实现及说明

```
void Worker::sort() {
  bool running;
  int avelen = (n + nprocs - 1) / nprocs;
  int nfull = n / avelen;
  int nlive = nfull + ((int)n != nfull * avelen);
  // nlive 表示 block_len 非零的进程数

  // data comm
  float *data_recv = new float[avelen + 5];
  float *data_swp = new float[block_len + 5];
  float *pdata = data, *swp = data_swp;
  int partner, diff;
  MPI_Request mpireq[30];
  // stopsig comm
  float comm_send[3], comm_recv[3];
  int next = (rank + 1) % nlive;
  int prev = (rank + nlive - 1) % nlive;


  // initialize
  {
    // 手写排序函数，提高效率
    uint *p = static_cast<uint*>(static_cast<void*>(&data[0]));
    uint *q = static_cast<uint*>(static_cast<void*>(&data_swp[0]));
    bucsort(p, q, block_len);
  }
  if (nlive == 1) return;
  // first round: (0 1) (2 3)
  if (rank & 1) diff = -1;
  else diff = 1;

  // 对于 `n` 较小的情况，进行多次通讯反而降低速度，因此强行使其跑满 `nproc` 次
  if (n <= 500000) {
    for (int g = 1; g <= nlive; g ++) {
      if (block_len == 0) break;
      partner = rank + diff;
      int recv_len = (partner < nfull ? avelen : (partner == nfull ? n - avelen * nfull : 0));
      running = (partner >= 0 && partner < nlive);
      if (running) {
        // 进程对互相传输数据，各自归并、取出所需的数据
        MPI_Isend(pdata, block_len, MPI_FLOAT, partner, 1, MPI_COMM_WORLD, &mpireq[0]);
        // 从 1 开始索引，减少一次判断（其实未必有优化）
        MPI_Irecv(data_recv + 1, recv_len, MPI_FLOAT, partner, 1, MPI_COMM_WORLD, &mpireq[1]);
        MPI_Waitall(2, mpireq, nullptr);
        // 哨兵
        data_recv[0] = -1.0/0.0;
        data_recv[recv_len + 1] = 1.0/0.0;

        // 进行归并排序
        if (diff > 0) {
          if (pdata[block_len - 1] > data_recv[1]) {
            for (int i = 0, u = 0, v = 1; i < (int)block_len; i ++) {
              if (pdata[u] <= data_recv[v])
                swp[i] = pdata[u ++];
              else
                swp[i] = data_recv[v ++];
```

```
          }
          std::swap(pdata, swp);
        }
      } else {
        // take out largest block_len floats
        if (data_recv[recv_len] > pdata[0]) {
          for (int i = block_len - 1, u = block_len - 1, v = recv_len; i >= 0; i --) {
            if (pdata[u] >= data_recv[v])
              swp[i] = pdata[u --];
            else
              swp[i] = data_recv[v --];
          }
          std::swap(pdata, swp);
        }
      }
    }
    diff = - diff;
  }
  if (pdata != (float*)data)
    memcpy(data, pdata, sizeof(float) * block_len);
  delete[] data_recv;
  delete[] data_swp;
  return; // haha
}

for (int g = 1; ; g ++) {
  if (nlive == 1) break;
  // skip sleeping process
  if (block_len == 0) break;
  // partner info
  partner = rank + diff;
  int recv_len = (partner < nfull ? avelen : (partner == nfull ? n - avelen * nfull : 0));
  running = (partner >= 0 && partner < nlive);
  // halt signal
  int steady = 0;

  if (running) {
    MPI_Isend(pdata, block_len, MPI_FLOAT, partner, 1, MPI_COMM_WORLD, &mpireq[0]);
    MPI_Irecv(data_recv + 1, recv_len, MPI_FLOAT, partner, 1, MPI_COMM_WORLD, &mpireq[1]);
    MPI_Waitall(2, mpireq, nullptr);
    data_recv[0] = -1.0/0.0;
    data_recv[recv_len + 1] = 1.0/0.0;

    // comm startup
    // 通过 ring_allreduce 来计算数列是否有序，第一轮先传递进程对的最大值进行比较
    int cntreq = 0;
    if (diff > 0 && rank > 0) {
      MPI_Irecv(comm_recv, 1, MPI_FLOAT, prev, 2, MPI_COMM_WORLD, &mpireq[0]);
      cntreq = 1;
    } else if (diff < 0 && rank + 1 < nlive) {
      comm_send[0] = std::max(pdata[block_len - 1], data_recv[recv_len]);
      MPI_Isend(comm_send, 1, MPI_FLOAT, next, 2, MPI_COMM_WORLD, &mpireq[0]);
      cntreq = 1;
    }

    // interval for comm to wake up
    // 每进行一段运算需要将先前的 MPI 函数回收一下，在此声明一个 interval 作为每次运算的大小
    int stage = 0;
    int interval = std::max((int)block_len / nlive, 1000);
```

```
if (diff > 0) {
  // |self, partner|, takes out smallest block_len floats
  int i = 0, u = 0, v = 1;
  while (i < (int)block_len) {
    int endpos = std::min(i + interval, (int)block_len);
    for (; i < endpos; i ++) {
      if (pdata[u] <= data_recv[v]) swp[i] = pdata[u ++];
      else                          swp[i] = data_recv[v ++];
    }
    // comm & calc async
    if (cntreq) {
      MPI_Waitall(cntreq, mpireq, nullptr);
      if (stage == 0) steady = swp[0] < comm_recv[0];       // inter process_pair checking
      else            steady |= (int)(comm_recv[0] + 1e-6); // ring allreduce
    }
    ++ stage;
    cntreq = 0;
    if (stage >= nlive) continue;
    // start next round
    comm_send[0] = steady;
    MPI_Isend(comm_send, 1, MPI_FLOAT, next, 2, MPI_COMM_WORLD, &mpireq[0]);
    MPI_Irecv(comm_recv, 1, MPI_FLOAT, prev, 2, MPI_COMM_WORLD, &mpireq[1]);
    cntreq = 2;
  }
  // deal the remaining
  while (stage < nlive) {
    if (cntreq) {
      MPI_Waitall(cntreq, mpireq, nullptr);
      if (stage == 0) steady = swp[0] < comm_recv[0]; // inter process_pair checking
      else            steady |= (int)(comm_recv[0] + 1e-6);
    }
    ++ stage;
    cntreq = 0;
    if (stage >= nlive) break;
    // start next phase
    comm_send[0] = steady;
    MPI_Isend(comm_send, 1, MPI_FLOAT, next, 2, MPI_COMM_WORLD, &mpireq[0]);
    MPI_Irecv(comm_recv, 1, MPI_FLOAT, prev, 2, MPI_COMM_WORLD, &mpireq[1]);
    cntreq = 2;
  }
} else {
  // diff < 0, take out largest block_len floats
  int i = block_len - 1, u = block_len - 1, v = recv_len;
  while (i >= 0) {
    int endpos = std::max(0, i - interval);
    for (; i >= endpos; i --) {
      if (pdata[u] >= data_recv[v]) swp[i] = pdata[u --];
      else                          swp[i] = data_recv[v --];
    }
    // comm & calc async
    if (cntreq) {
      MPI_Waitall(cntreq, mpireq, nullptr);
      if (stage != 0) steady |= (int)(comm_recv[0] + 1e-6);
    }
    ++ stage;
    cntreq = 0;
    if (stage >= nlive) continue;
    // start next round
    comm_send[0] = steady;
    MPI_Isend(comm_send, 1, MPI_FLOAT, next, 2, MPI_COMM_WORLD, &mpireq[0]);
```

```
          MPI_Irecv(comm_recv, 1, MPI_FLOAT, prev, 2, MPI_COMM_WORLD, &mpireq[1]);
          cntreq = 2;
        }
        // deal the remaining
        while (stage < nlive) {
          if (cntreq) {
            MPI_Waitall(cntreq, mpireq, nullptr);
            if (stage != 0) steady |= (int)(comm_recv[0] + 1e-6);
          }
          ++ stage;
          cntreq = 0;
          if (stage >= nlive) break;
          // start next phase
          comm_send[0] = steady;
          MPI_Isend(comm_send, 1, MPI_FLOAT, next, 2, MPI_COMM_WORLD, &mpireq[0]);
          MPI_Irecv(comm_recv, 1, MPI_FLOAT, prev, 2, MPI_COMM_WORLD, &mpireq[1]);
          cntreq = 2;
        }
      }
      std::swap(pdata, swp);
    } else {
      // 可能有 1 至 2 个进程不参与这一轮排序，他们需要进行 allreduce
      if (rank == 0) {
        comm_send[0] = pdata[block_len - 1];
        MPI_Isend(comm_send, 1, MPI_FLOAT, next, 2, MPI_COMM_WORLD, &mpireq[0]);
        MPI_Waitall(1, mpireq, nullptr);
      } else {
        MPI_Irecv(comm_recv, 1, MPI_FLOAT, prev, 2, MPI_COMM_WORLD, &mpireq[0]);
        MPI_Waitall(1, mpireq, nullptr);
        steady = comm_recv[0] > pdata[0];
      }
      for (int u = 1; u < nlive; u ++) {
        comm_send[0] = steady;
        MPI_Isend(comm_send, 1, MPI_FLOAT, next, 2, MPI_COMM_WORLD, &mpireq[0]);
        MPI_Irecv(comm_recv, 1, MPI_FLOAT, prev, 2, MPI_COMM_WORLD, &mpireq[1]);
        MPI_Waitall(2, mpireq, nullptr);
        steady |= (int)(comm_recv[0] + 1e-6);
      }
    }
    if (!steady) break;
    diff = - diff;
  }
  if (pdata != (float*)data) {
    memcpy(data, pdata, sizeof(float) * block_len);
  }
  delete[] data_recv;
  delete[] data_swp;
}
```

主要优化

首先可以意识到最主要的优化是将传输与计算异步进行，以及降低初始化的耗时

1. 利用浮点数的表示方法，手动实现一个桶排序，考虑到 L1 cache 的大小，将桶的大小选取为 2^8，这一优化使得初始排序的时间减少了 400ms
2. 通过测试，决定对于 $n \leq 5e5$ 的情况，不交流终止时间，在 n 较小的情况下，这能快出大约 6ms
3. 采用 ring_allreduce 实现终止时间的交流（对于混乱程度不高的数据，运行效率能够大大提升），并将其与归并计算重叠进行（理论上肯定有优化，但可能被测量误差掩盖了）

此外，我还曾试图将数据分块进行传输，但是注意到这反而拖慢了运行速度。通过 OSU 测试发现进程通信的带宽极高，可能是数

据分块后，多次调用 MPI 造成的的花销，超过传输与计算并行带来的收益。

加速比

nprocs	耗时 (ms)	加速比
1×1	2960	1
1×2	1839	1.610
1×4	1169	2.532
1×8	910	3.253
1×16	694.6	4.261
2×16	591.3	5.007

附：桶排序代码

```
typedef unsigned int uint;

static void _sort(uint *a, uint *swp, int n) {
    int buc0[1 << 8], buc1[1 << 8], buc2[1 << 8], buc3[1 << 8];
    memset(buc0, 0, sizeof buc0);
    memset(buc1, 0, sizeof buc0);
    memset(buc2, 0, sizeof buc0);
    memset(buc3, 0, sizeof buc0);
    for (int i = 0; i < n; ++ i) {
        ++ buc0[a[i]       & 255];
        ++ buc1[a[i] >>  8 & 255];
        ++ buc2[a[i] >> 16 & 255];
        ++ buc3[a[i] >> 24 & 255];
    }
    for (int i = 1; i < 256; ++ i) {
        buc0[i] += buc0[i - 1];
        buc1[i] += buc1[i - 1];
        buc2[i] += buc2[i - 1];
        buc3[i] += buc3[i - 1];
    }
    // round 1
    uint *ptr = a + n - 1;
    for (int iter = n >> 3; iter; iter --) {
        swp[-- buc0[ptr[ 0] & 255]] = ptr[ 0];
        swp[-- buc0[ptr[-1] & 255]] = ptr[-1];
        swp[-- buc0[ptr[-2] & 255]] = ptr[-2];
        swp[-- buc0[ptr[-3] & 255]] = ptr[-3];
        swp[-- buc0[ptr[-4] & 255]] = ptr[-4];
        swp[-- buc0[ptr[-5] & 255]] = ptr[-5];
        swp[-- buc0[ptr[-6] & 255]] = ptr[-6];
        swp[-- buc0[ptr[-7] & 255]] = ptr[-7];
        ptr -= 8;
    }
    while (ptr >= a) {
        swp[-- buc0[ptr[0] & 255]] = ptr[0];
        ptr --;
    }
    // round 2
    ptr = swp + n - 1;
    for (int iter = n >> 3; iter; iter --) {
        a[-- buc1[ptr[ 0] >> 8 & 255]] = ptr[ 0];
        a[-- buc1[ptr[-1] >> 8 & 255]] = ptr[-1];
        a[-- buc1[ptr[-2] >> 8 & 255]] = ptr[-2];
        a[-- buc1[ptr[-3] >> 8 & 255]] = ptr[-3];
        a[-- buc1[ptr[-4] >> 8 & 255]] = ptr[-4];
```

```
        a[-- buc1[ptr[-5] >> 8 & 255]] = ptr[-5];
        a[-- buc1[ptr[-6] >> 8 & 255]] = ptr[-6];
        a[-- buc1[ptr[-7] >> 8 & 255]] = ptr[-7];
        ptr -= 8;
    }
    while (ptr >= swp) {
        a[-- buc1[ptr[0] >> 8 & 255]] = ptr[0];
        ptr --;
    }
    // round 3
    ptr = a + n - 1;
    for (int iter = n >> 3; iter; iter --) {
        swp[-- buc2[ptr[ 0] >> 16 & 255]] = ptr[ 0];
        swp[-- buc2[ptr[-1] >> 16 & 255]] = ptr[-1];
        swp[-- buc2[ptr[-2] >> 16 & 255]] = ptr[-2];
        swp[-- buc2[ptr[-3] >> 16 & 255]] = ptr[-3];
        swp[-- buc2[ptr[-4] >> 16 & 255]] = ptr[-4];
        swp[-- buc2[ptr[-5] >> 16 & 255]] = ptr[-5];
        swp[-- buc2[ptr[-6] >> 16 & 255]] = ptr[-6];
        swp[-- buc2[ptr[-7] >> 16 & 255]] = ptr[-7];
        ptr -= 8;
    }
    while (ptr >= a) {
        swp[-- buc2[ptr[0] >> 16 & 255]] = ptr[0];
        ptr --;
    }
    // round 4
    ptr = swp + n - 1;
    for (int iter = n >> 3; iter; iter --) {
        a[-- buc3[ptr[ 0] >> 24 & 255]] = ptr[ 0];
        a[-- buc3[ptr[-1] >> 24 & 255]] = ptr[-1];
        a[-- buc3[ptr[-2] >> 24 & 255]] = ptr[-2];
        a[-- buc3[ptr[-3] >> 24 & 255]] = ptr[-3];
        a[-- buc3[ptr[-4] >> 24 & 255]] = ptr[-4];
        a[-- buc3[ptr[-5] >> 24 & 255]] = ptr[-5];
        a[-- buc3[ptr[-6] >> 24 & 255]] = ptr[-6];
        a[-- buc3[ptr[-7] >> 24 & 255]] = ptr[-7];
        ptr -= 8;
    }
    while (ptr >= swp) {
        a[-- buc3[ptr[0] >> 24 & 255]] = ptr[0];
        ptr --;
    }
}

static void bucsort(uint *a, uint *swp, int n) {
    if (n <= 1) return;
    int l = 0, r = n - 1;
    while (l < r) {
        while (l < r && (a[l] & (1u << 31))) l ++;
        while (l < r && (a[r] & (1u << 31)) == 0) r --;
        if (l < r) {
            std::swap(a[l], a[r]);
        } else break;
    }
    if (l >= 0 && (a[l] & (1u << 31)) == 0) l --;
    if (l > 0) {
        for (int i = 0; i <= l; i ++) a[i] ^= (~0);
        _sort(a, swp, l + 1);
        for (int i = 0; i <= l; i ++) a[i] ^= (~0);
```

```
    }
    if (r < n - 1 && (a[r] & (1u << 31))) r ++;
    if (r < n - 1) _sort(a + r, swp, n - r);
}
```