```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <mpi.h>

#include "Histogram.h"
#include "File_IO.h"
#include "Terminal_IO.h"
#include "Sort.h"
#include "Communication.h"

int getIndexOfNode(int node, short int *activeNodes, short int size);
int getSuccessorOfNode(int pNr, short int *activeNodes, short int size);
int getPredecessorOfNode(int pNr, short int *activeNodes, short int size);
short int* deleteOddProcessNumber(short int *activeNodes, short int *activeNodes
_size);
char* my_itoa(int wert, int laenge);

Histogram** initHistogramArray(Histogram *data, unsigned int *size) {
  // Erstelle ein Array mit Referenzen (Pointern) auf die Daten
    Histogram **ref_data = (Histogram**) malloc (sizeof(Histogram*)*(*size));

    unsigned int n;
    for (n = 0; n < (*size); n++) { // kopiere die Adressen in das neue Array.
        ref_data[n] = &data[n];
    }

    return ref_data;
}

int main (int argc, char *argv[]) {
    int myRank;
    int ranks;

    // init mpi
    MPI_Init(&argc, &argv);

    // get rank of this prozess
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    // get number of prozesses
    MPI_Comm_size(MPI_COMM_WORLD, &ranks);



    MPI_Datatype HISTOGRAM_TYPE, oldtypes[2];
  int          blockcounts[2];
  // MPI_Aint type used to be consistent with syntax of
  // MPI_Type_extent routine
  MPI_Aint    offsets[2], extent;

  // Setup description of the 4 MPI_FLOAT fields x, y, z, velocity
  offsets[0] = 0;
  oldtypes[0] = MPI_UNSIGNED_CHAR;
  blockcounts[0] = 52;

  // Setup description of the 2 MPI_INT fields n, type
  // Need to first figure offset by getting size of MPI_FLOAT
  MPI_Type_extent(MPI_UNSIGNED_CHAR, &extent);
  offsets[1] = 52 * extent;
  oldtypes[1] = MPI_INT;
```



```c
  blockcounts[1] = 1;

  // Now define structured type and commit it
  MPI_Type_struct(2, blockcounts, offsets, oldtypes, &HISTOGRAM_TYPE);
  MPI_Type_commit(&HISTOGRAM_TYPE);

    Histogram *data = NULL;
    unsigned int size_data = 0;


  #ifdef Zeitmessung
  // time variables
    double startTime = 0, endTime = 0, timeUsed = 0, totalTime = 0;

    if (myRank == 0) {
      startTime = MPI_Wtime(); // set start time
      totalTime = startTime;
  }
  #endif // Zeitmessung

  const char* filename = "/usr/local/sortMe.txt";
  //const char* filename = "sortMe_1000.txt";
    // Lese Datei und bekomme das die Histogramme zurÃ¼ck.
    data = readFile(filename, myRank, ranks, data, &size_data);

  #ifdef Zeitmessung
  if (myRank == 0) {
    endTime = MPI_Wtime();
      timeUsed = endTime - startTime;
    printf("time used to read file: %s = %lf \n", filename, timeUsed);
    startTime = MPI_Wtime();
  }
  #endif

  Histogram **ref_data = initHistogramArray(data, &size_data);

    // Hier haben wir nun das Histogram dieses Prozesses.

    // Das Histogram soll nun sortiert werden.
    ref_data = sort(ref_data, &size_data); // FEHLER IN SORT: Informationen gehe
n verloren

  #ifdef Zeitmessung
  if (myRank == 0) {
    endTime = MPI_Wtime();
      timeUsed = endTime - startTime;
    printf("time used to sort file local: %s = %lf \n", filename, timeUsed);
    startTime = MPI_Wtime();
  }
  #endif

  // h ist nun sortiert.
  // also sollte der inhalt von h mal asugegeben werden
  //printHistogramArray(mixed, size);
    //writeFile("out.txt",filename, mixed, &size);

  /* Hier beginnt der MPI SpaÃM-^_ */
  // Jeder Knoten hat seine Elemente die er aus dem File gelesen hat sortiert un
d kann diese nun an andere Knoten senden.
  short int *activeNodes = (short int*) malloc (sizeof(short int)*ranks);
  int i;
  for (i = 0; i < ranks; i++) {
```

```
    activeNodes[i] = i;
  }

  // Anzahl der Elemente
  short int activeNodes_size = ranks;
  /*
  #ifdef Zeitmessung
  if (myRank == 0) {
    startTime = MPI_Wtime();
    printf("Schreibe Daten ... ");
    if (myRank == 0) writeFileFromMemory("/tmp/out.txt", filename, ref_data, &si
ze_data);
    endTime = MPI_Wtime();
      timeUsed = endTime - startTime;
    printf("DONE, time used: %lf\n",timeUsed);

  }
  #endif
  */

  //Histogram **sorted_merged_Histogram = ref_data;

  // solange wie mehr als 1 Element im activeNodes Array vorhanden ist
  while (activeNodes_size > 1) {
    // jetzt mÃ¼ssen wir bestimmen ob dieser Prozess empfangen oder senden soll?
      // suche diese Prozessnummer in den activeNodes und gib den Index zurÃ¼ck
    int indexOfThisNodeInArray = getIndexOfNode(myRank, activeNodes, activeNodes
_size);

    // wenn Index gerade dann empfange vom Nachfolger
      if (indexOfThisNodeInArray % 2 == 0) {
        // Jetzt mÃ¼ssen wir raus finden wer der Nachfolger ist
      int successor = getSuccessorOfNode(myRank, activeNodes, activeNodes_size);

      if (successor != -1) { // wenn es einen nachfolger gibt empfange und merge
        //printf("Prozess: %d empfaengt von Prozess: %d \n", myRank, successor);
        unsigned int size_received; // Speichert die Anzahl der Elemente im Hist
ogram

        //_printHistogramArray(data, size_data); // Zeigt Daten im Speicher (uns
ortiert): OK
        //printHistogramArray(ref_data, size_data); // lokale daten vor empfang
(soritert): OK!
        // Empfange Datan, data wird entprechend erweitert
        data = receiveHistogram(successor, &size_received, data, size_data, &HIS
TOGRAM_TYPE, ref_data);
        //_printHistogramArray(data, size_data); // Zeigt Daten im Speicher (sor
tiert): OK !
        //printHistogramArray(ref_data, size_data); // lokale daten nach empfang
: OK!
        // VergrÃ¶Ã–^_ere Data
        //data = (Histogram*) realloc (data, sizeof(Histogram) * (size_data + si
ze_received));

        // kopiere received Elemente in diesen Speicher
        //memcpy(data+size_data, data_received, size_received * sizeof(Histogram
));

        // Gebe Alten Speicher frei
        //free(data_received);

                // Referenz auf die empfangenen Daten, fÃ¼r die sortierung.
```



```
        Histogram **ref_received_data = initHistogramArray((data+size_data), &si
ze_received);
        //printHistogramArray(ref_received_data, size_received); // Referenz auf
 empfangene Daten: OK!

        // Neuinitalisierung der lokalen referenzen falls diese ungÃ¼ltig geword
en sind SCHLECHTE IDEE, da somit die referenzen auf die ursprÃ¼nglich unsortiert
en DAten zeigen und nicht auf die bereits sortierten
        ref_data = initHistogramArray(data, &size_data);
        //printHistogramArray(ref_data, size_data);  // Referenzen auf die local
en Daten: OK!

                // In diesem Speicherbereich kommen die Referenzen beider
        Histogram **sorted = (Histogram**) malloc (sizeof(Histogram*)*(size_data
+size_received));

        // do merge
          sorted = merge(ref_data, &size_data, ref_received_data, &size_received
,  sorted);
          //printHistogramArray(sorted, size_data+size_received); // Referenzen
auf sortierte daten Lokal + Received: WIEDER OK! ABER NICHT mit 10000 Zeilen!

          // ref_data zeigt nun auf die sortierten referenzen
          ref_data = sorted;
          size_data += size_received; // merke die neue anzahl der elemente

          //printHistogramArray(ref_data, size_data);
      }

      // in jedem fall lÃ¶sche knoten, die gesendet haben aus liste
      // LÃ¶sche alle nodes aus liste mit ungeradem index
      short int *rest_of_activeNodes = deleteOddProcessNumber(activeNodes, &acti
veNodes_size);
      free(activeNodes);
      activeNodes = rest_of_activeNodes; // Hier stehen nun die restlichen activ
en Knoten (Prozesse) drinn.
      }

      // wenn Index ungerade dann sende an VorgÃ¤nger
      if (indexOfThisNodeInArray % 2 != 0) {
        // sende an VorgÃ¤nger
      int predecessor = getPredecessorOfNode(myRank, activeNodes, activeNodes_si
ze);
      //printf("Prozess: %d sendet an Prozess: %d\n", myRank, predecessor);
      //printHistogramArray(ref_data, size_data);
      sendHistogram(predecessor, ref_data, size_data, &HISTOGRAM_TYPE);

      //printf("Prozess: %d hat gesendet\n",myRank);



      //printf("MPI_Finalize() Prozess: %d \n", myRank);

      free(data);
      //data = (Histogram*) malloc (sizeof(Histogram));
      //size_data = 0;
      /*
      unsigned int size_received;
      // empfange teil der soriterten Daten (von Knoten Null) um diese partiel a
uf platte zu schreiben
      data = receiveSortedHistogram(0, &size_received, &HISTOGRAM_TYPE);
      Histogram **ref_data = initHistogramArray(data, &size_received);
```

```
    //free(data);
    //data = (Histogram*) malloc (sizeof(Histogram));


    char buffer[15] = {'/','t','m','p','/','o','u','t'};
    char *myrank = my_itoa(myRank, 2);
    //printf("%s\n",myrank);
    strncat(buffer,myrank,2);
    strncat(buffer,".txt",4);
    printf("%s\n",buffer);

    #ifdef Zeitmessung
    startTime = MPI_Wtime();
    #endif

    writeFile(buffer, filename, ref_data, &size_received); // Klaartext
    //writeFileFromMemory(buffer, filename, ref_data, &size_received);

    #ifdef Zeitmessung
    endTime = MPI_Wtime();
      timeUsed = endTime - startTime;
    printf("Prozess: %d DONE, time used: %lf\n",myRank, timeUsed);
    #endif

    free(ref_data); // Array mit Pointern auf Histogramme
          free(data); // Original
    */
    // DONE!
    MPI_Finalize();
    return EXIT_SUCCESS;


    }

}



if (myRank == 0) {
  /*int node;
  for (node = 1; node < ranks; node++) {
    sendSortedHistogram(node, ranks, data, size_data, &HISTOGRAM_TYPE);
  }


  #ifdef Zeitmessung
    startTime = MPI_Wtime();
  #endif

  //printHistogramArray(ref_data, size_data);

  char buffer[15] = {'/','t','m','p','/','o','u','t'};
  char *myrank = my_itoa(myRank, 2);
  //printf("%s\n",myrank);
  strncat(buffer,myrank,2);
  strncat(buffer,".txt",4);
  printf("%s\n",buffer);
  unsigned int size_p0 = size_data/ranks;
  printf("Schreibe In Datei\n");
  writeFile(buffer, filename, ref_data, &size_p0); // Klaartext
  //writeFile("/tmp/out.txt", filename, ref_data, &size_data); // Klaartext
```



```
    //writeFileFromMemory(buffer, filename, ref_data, &size_data);

    #ifdef Zeitmessung
      endTime = MPI_Wtime();
        timeUsed = endTime - startTime;
      printf("Prozess 0 DONE, time used: %lf\n",timeUsed);
    #endif
    */


    printControlLines(ref_data, filename, 545146);

    int index;
    for (index = 10; index <= 10000000; index*=10) {
      printControlLines(ref_data, filename, index);
    }

    free(ref_data); // Array mit Pointern auf Histogramme

  }
    #ifdef Zeitmessung
    endTime = MPI_Wtime();
      timeUsed = endTime - totalTime;

    printf("%d elements in array!\n",size_data);
    printf("time from start to very end = %lf \n", timeUsed);
    #endif

  MPI_Type_free(&HISTOGRAM_TYPE);

    MPI_Finalize();
    return EXIT_SUCCESS;
}

short int* deleteOddProcessNumber(short int *activeNodes, short int *activeNodes
_size) {
  int i;
  short int *rest_of_activeNodes = (short int*) malloc (sizeof(short int));
  int size = 0;
  for (i = 0; i < *activeNodes_size; i++) {
    if (i % 2 == 0) {
      // kopiere Elemente an geradem Index in neues Array;
      size++;
      rest_of_activeNodes = realloc(rest_of_activeNodes, sizeof(short int)*size)
;
      int index_activeNodes = size-1;
      rest_of_activeNodes[index_activeNodes] = activeNodes[i];
    }
  }
  *activeNodes_size = size;
  return rest_of_activeNodes;
}

/**
 * Get the index of the given node in vector.
 * @param node
 * @param activeNodes
 * @return int The Index of the node in the vector.
 */
int getIndexOfNode(int pNr, short int *activeNodes, short int size) {
    int index;
    for (index = 0; index < size ; index++) {
```

```
        if (pNr == activeNodes[index]) { // Wenn die gesuchte Prozessnummer gefu
nden wurde
            return index; // Gebe den Index, an welcher dieser Prozess im Array
steht.
        }
    }
    return -1;
}

/**
 * Get the successor of the given node from vector.
 * @param node
 * @param activeNodes A Pointer of
 * @return unsigned char A pointer to the successor node.
 */
int getSuccessorOfNode(int pNr, short int *activeNodes, short int size) {
    int i;
    for (i = 0; i < size; i++) {
        if (pNr == activeNodes[i]) {
            // get Successor
            if (i == size-1) { // wenn i am letzten index
                // dann gibt es keinen nachfolger
                return -1;
            }
            return activeNodes[i+1];
        }
    }
    return -1;
}

/**
 * Get the Predecessor of the given node from vector.
 * @param node
 * @param activeNodes A Pointer of
 * @return unsigned char A pointer to the Predecessor node.
 */
int getPredecessorOfNode(int pNr, short int *activeNodes, short int size) {
    int i;
    for (i = 0; i < size; i++) {
        if (pNr == activeNodes[i]) {
            // get Predecessor
            return activeNodes[i-1];
        }
    }

    return -1;
}

char* my_itoa(int wert, int laenge) {
  char *ret = (char*) malloc ((laenge+1) * sizeof(char));
  int i;

  for (i = 0; i < laenge; i++) {
    ret[laenge-i-1] = (wert % 10) + 48;
    wert = wert / 10;
  }

  ret[laenge]='\0';
  return ret;
}
```