Parallel and Distributed Programming
Assignment 2

Daniel Salvador
Peili Guo

May 15, 2019

Introduction

The goal of this assignment is implement parallel quick sort using MPI with three different pivot strategy when communicating with different processors.

Quick sort is a good algorithm for sorting numbers with average time complexity $O(nlog(n))$. The serial quick sort is to choose a pivot element, and do partition, to divide the numbers into two sub-arrays, one that is smaller than the pivot, and one that is larger than the pivot. And repeat partition on the two sub-arrays individually, until the size of the parts are 1, where partition is not carried out further. The array is then sorted.

The alogotirhm for parallel quick sort that we implement in this report is described below where p is the number of proccessors:

1. Read the input file to dynamic allocated memory.
2. Divided the data into p equal parts, and each process take one part and do serial quick sort.
3. Now each process has a sorted array and will perform mpi_qsort with the following steps:
   (a) Select pivot element within each process set
   (b) Divide the data into two sub sets according to the pivot element locally in each process.
   (c) Split the processes into two groups and exchange data pairwise between them so that one group get data less than the pivot and the other get data larger than pivot.
   (d) Locally merge the two sets on each process
4. repeat Step 3 repeated until each group consists of one single process.
5. Put together the list one after another by the order of the rank of process in the global commmunicator and we get a sorted list.

For the local serial quick sort within each process in step 2, the pivot is choosen as the number that is in the middle of the array. There are three different strategy for choosing the pivot element in the mpi_qsort:

1. Select the median value in the process with rank = 0 in each group of processors.
2. Selection the median value of all medians in each processor group.
3. Select the mean value of all medians in each processor group.

We use MPI to handle the communication between the process.

1 Solution Design

The implementation of the serial quick sort is straightforward, but the parallel mpi_qsort is not very straightforward.
We follow the algorithm in the introduction part. Each process read the input txt file into dynamically located arr and compute its index of start and stop and then copy the part to local array to do local quick sort. We put a barrier to wait for each process to finish local quick sort before mpi_qsort. The mpi_qsort is a recursive function that return the pointer to local array:

```
int* mpi_qsort(int* data, int len, MPI_Comm com, int option)
```

1. If the size of the communicator is 1, we send the current local array to process 0 for collection of final result and return the local array that is sorted.
2. Compute the group pivot, and broadcast to the group processor.
3. if the rank of process is smaller than size/2 in the communicator, we divide the local array into two parts: one smaller than the pivot and one larger than the pivot. We keep the smaller part, and send the larger part to process with rank = size/2+1.
   if the rank is larger or equal to size/2, we divide the local array into two parts: one smaller than the pivot and one larger than the pivot. We keep the larger part and send the smaller part.
4. Split the communicator into two halves.
5. call the mpi_qsort again with the new sorted local array, new length, and new communicator.

For the communication, We use MPI_Probe and MPI_Get_count to get the size of receiving array and use MPI_Isend and MPI_Recv to exchange either the smaller or larger part of the local array in the group commuicator.

2 Results

We experiment with the number of processor in 1 2 4 8 16 on `Rackham` cluster. All the timings are taken using `MPI_Wtime()` from doing local quick sort till when the sorted array has finished collecting results from other processors. For numerical experiment, we test large input files with size of 125 000 000, 500 000 000, and 1 000 000 000.

size(000000)	number of processor	pivot1	pivot2	pivot3
125	1	16.83	16.76	16.87
125	2	9.15	9.18	9.14
125	4	5.38	5.41	5.41
125	8	3.25	3.07	3.10
125	16	1.80	2.03	2.22
reverse125	1	5.20	5.24	5.35
reverse125	2	3.68	3.32	3.54
reverse125	4	2.59	2.33	2.41
reverse125	8	2.80	1.56	1.53
reverse125	16	1.18	1.15	1.11
500	1	75.10	73.26	73.40
500	2	38.09	38.64	38.14
500	4	20.95	22.60	22.72
500	8	14.44	13.03	14.39
500	16	7.40	8.99	10.65
1000	1	154.05	154.68	154.61
1000	2	78.67	78.85	79.31
1000	4	46.82	47.05	46.95
1000	8	24.07	26.79	27.04
1000	16	16.15	16.25	17.40

Table 1: Execution time for running different size of array on different processors, pivot1, pivot2, and pivot3 are the strategy choosing pivot element for mpi_qsort, details in introduction section.

3 Conclusions

We successfully implement the parallel quick sort using MPI and we can observe speed up when running on multi cores.

The average quick sort has time complexity $O(nlog(n))$, we can do a simple analysis on the parallel quick sort, that the local quick sort will have the complexity of $O(n/p \times log(n/p)$, where p is the number of cores assigned, and the communication is of $O(n)$, as we need to go through the array to find the pivot and split, and exchange data, and merge.

The experiment with array of size 125000000, 500000000,1000000000, and reverse sorted 125000000 shows a max of around 10x speed up when running on 16 cores, and 6x speed up when running on 8 cores in Figure 1. If assigning more cores for parallel quick sort, we are likely to observe that the speed up will not increase as fast as as we increase the number of cores assigned. The reason is the communication overhead: the process of create new communication group, split the data into two subsets, exchange the data between processor, merge, all these will get more and more expensive as the number of cores increase.

From the speed up results in Figure 1 and 2, we cannot conclude on which pivot strategy is better. In general, the strategy of choosing the median value of the process 0 perform ok, but when experimenting reverse sorted array of size 125000000 on 8 cores, the speed up is only 2x, much lower compared with the other two strategies. Strategy 3 seems not performing very well, but then the difference among the three different strategies are not significant enough to draw statistic conclusions.

When we take a look at Figure 1, it seems that parallel quick sort is not efficient in solving reserve sorted data. The max speed up is around 4 times when running on 16 cores. However, if we take a look at Figure 3, the exact execution time is much less than the random array of the same size, as the local quick sort is probably the key in that case. The reason is that we make the local quick sort to choose the middle number as pivot, so that for a reverse sorted array it is optimised, as the array is always divided into two equal part. In the early trys of the implementation, the pivot was set to the last element, and the code cannot finish local quick sort at time out (the worst case for quick sort of $O(n^2)$. The motivation of the change is that we can see the effect of parallel quick sort, compare the speed up for an optimised serial local quick sort, and observe the communication overhead.

In Figure 2, we see the weak scalablity, where we keep n/p a constant (the workload for each process is constant). We see a rapid decrease in the efficiency from 1 core to 4 core, to around 75%, and further decreasing to around 70%. The same can observed in the same figure with circle marker that it decrease rapidly from 2 cores to 8 cores.

In all, the implementation is successful, we do expect heavy communication overhead and decrease in efficiency when increase the number of cores. It is worth pointing out that memory soon become an issue with all the dynamic allocated memory in the process. Sometimes, we have 2 or 3 times of the array size in local process. On top of that, the read input file and write output file take significant long time compare to the actual parallel quick sort. In the end, we would like to recommend: a change to binary files for data i/o and better management of all the dynamic allocated memory to avoid running out of memory on the parallel system should be implemented.

4 Appendix

```
Rackham.uppmax.uu.se properties:

Model name: Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
Available cores: 40
```

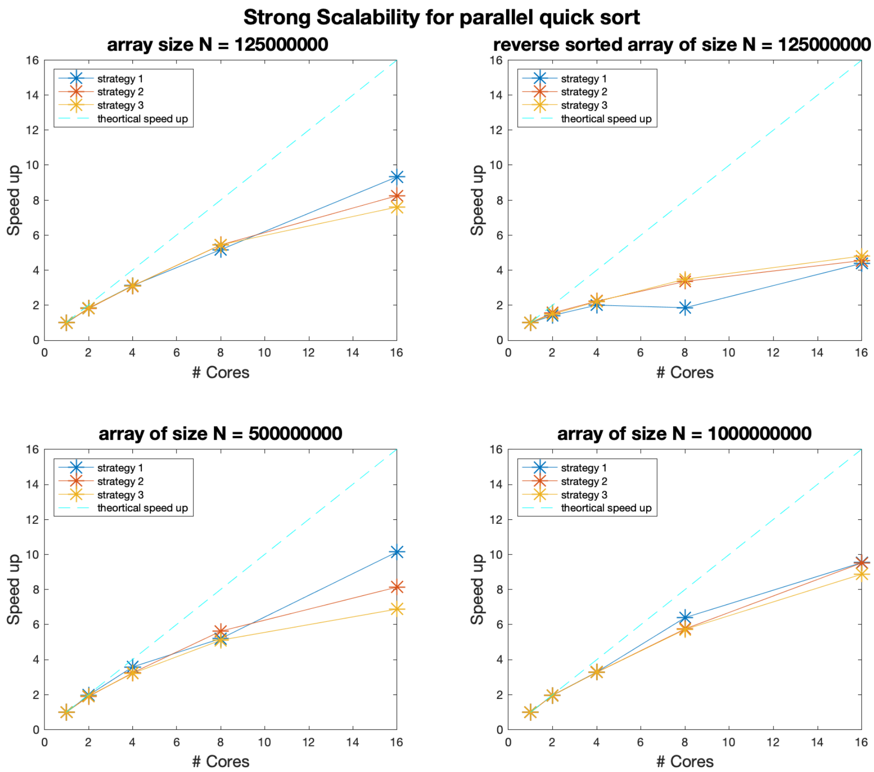

Figure 1: Strong scalability of parallel quick sort (Speedup) with different number of cores.

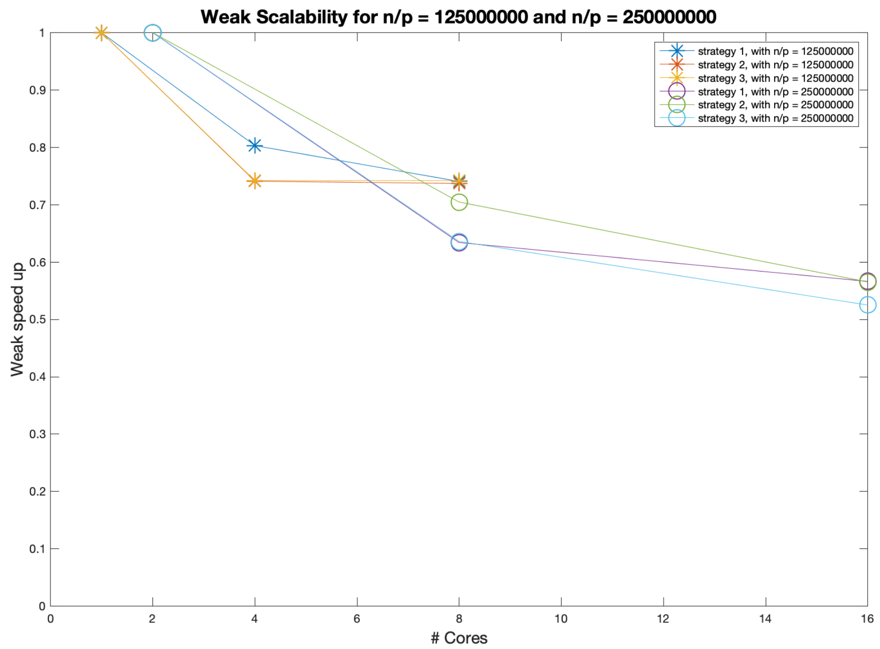

Figure 2: Weak scalability of parallel quick sort where we keep n/p constant.

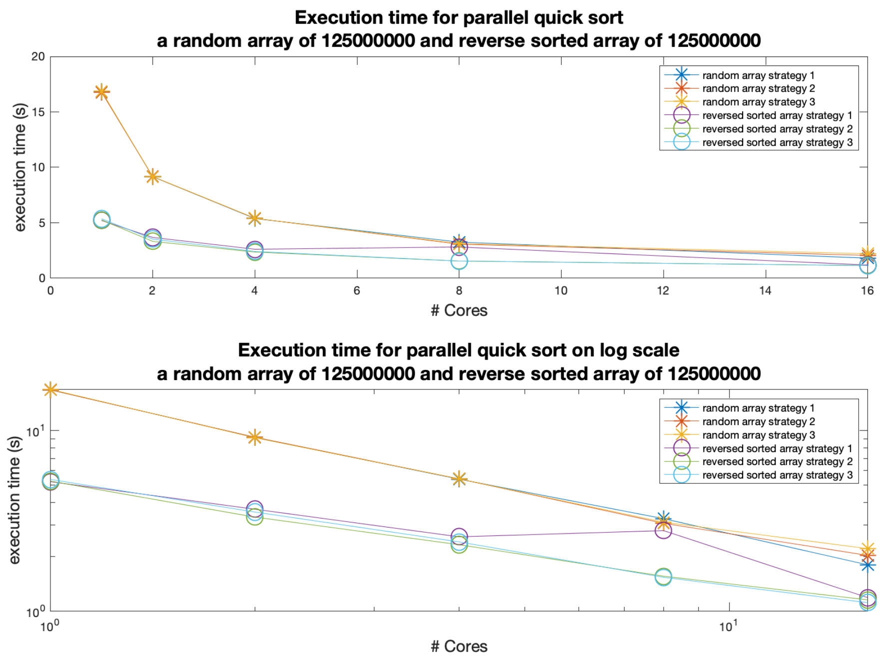

Figure 3: Comparison of execution time with array of size 125000000 one is random array and the other reverse sorted array.

5 Source Code

```c
/*****************************************************************************
* Run by typing: "mpirun -np p ./quicksort inputfile outputfile pivot_strategy"*
* p: Number of processors (square number)                                    *
* example:                                                                   *
* mpirun -np 4 ./quicksort input10.txt output10.txt 1                        *
*****************************************************************************/

#include <math.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//function called:
void print_array(int* a, int length);
int check_result(int *arr, int length);
void local_merge(int size1, int size2, int* arr1, int* arr2, int* c);
int partition(int* arr, int left, int right, int option);
void quicksort(int* arr, int left, int right, int option);
void copyArray(int *d1, int *d2, int start, int length);
int read_file(char *name, int** pp);
void save_result(char* name, int* arr, int n);
int* mpi_qsort(int* data, int len, MPI_Comm com, int option);


int main(int argc, char *argv[]){
  // set up MPI
  MPI_Init(&argc, &argv); //initialize
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); //get my number
  MPI_Comm_size(MPI_COMM_WORLD, &size); //get number of processors

  MPI_Status status;
  // Program arguments
  if(argc != 4){
    printf("please enter 3 input; to run ./qsort inputfile outputfile methods\n.");
    return -1;
  }
  // printf("input file: %s\n",argv[0]);
  // printf("pivot strategy %s\n", argv[3]);
  char* input_file = argv[1];
  char* output_file = argv[2];
  int option = atoi(argv[3]); //strategy number
  // printf("%s\n",input_file);
  int* arr;
  int n2;
  n2 = read_file(input_file, &arr);
  //*****************************************
  //read file and get n2, and chop it locally according to need.


  int chunk;              /* This many iterations will I do */
```

```
  int istart, istop;  /* Variables for the local loop   */

chunk  = n2/size;       /* Number of intervals per processor */
istart = rank*chunk;          /* Calculate start and stop indices  */
istop  = (rank+1)*chunk-1;      /* for the local loop               */

if (rank == size-1 ) {  istop = n2-1; } /* Make sure the last processor gets all the rest      */
int local_size;
local_size = istop - istart + 1;
int* local_arr;
local_arr = (int*)malloc(local_size*sizeof(int));
int local_index = 0;
for(int i = istart; i<=istop; i++){
  local_arr[local_index] = arr[i];
  local_index++;
}
free(arr);

//************************************************
//now we do local quick sort. and start the clock.
double t;
if(rank == 0) {t = MPI_Wtime ();}

quicksort(local_arr,0,local_size-1,1); //local quick sort
// print_array(local_arr, local_size);
//local sorted successfully.
MPI_Barrier(MPI_COMM_WORLD);  //wait for everyone to finish quicksort
local_arr = mpi_qsort(local_arr, local_size, MPI_COMM_WORLD,option);
//and switch switch switch swtich with mpi_qsort and ready to merge.

//*************************************************
// now we collect everything on process 0.
int k=0;
int num_get=0;
int num_tmp;
int* sorted_array;

int result=0;
if(rank==0){
  sorted_array = (int*)malloc(n2*sizeof(int));
  while(k<size){
    MPI_Probe(k, 444, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_INT, &num_tmp);
    // printf("num_tmp: %d", num_tmp);
    MPI_Recv(&sorted_array[num_get],num_tmp, MPI_INT, k, 444, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    num_get = num_get+ num_tmp;
    k++;
  }

  t = MPI_Wtime () -t ; //stop the clock
  result = check_result(sorted_array,n2); //check if result is correct
  printf("%.8f\n", t);
  // printf("%d, %.8f, %d, %d, %d \n", n2, t, size, result,
```

```
        option);
    /*
    FILE * fp;
    fp = fopen ("input1000.txt","a");
    fprintf (fp, "%d, %.8f, %d, %d, %d \n", n2, t, size,
        result, option);
    fclose(fp);
    */
    save_result(output_file, sorted_array, n2);
    free(sorted_array);
  }

  MPI_Barrier(MPI_COMM_WORLD);
  free(local_arr);

  MPI_Finalize(); /* Shut down and clean up MPI */
  return 0;
}




/************************************************************
 * function used
 */

void print_array(int* a, int length){
  printf("printing array below:\n");
  for(int i=0; i<length; i++){
    printf("%d. %d\n", i, a[i]);
  }
  printf("\nfinished\n");
}

int check_result(int *arr, int length){
    int i;
    for (i = 1; i < length; i++){
        if (arr[i - 1] > arr[i]){
            printf("error: a[%d] = %d, a[%d] = %d\n", i-1,
                arr[i-1], i, arr[i]);
            return -1;
        }
    }
    //printf("result correct\n");
    return 1;
}

void local_merge(int size1, int size2, int* arr1, int* arr2,
    int* c){//allocate c before
  int i=0;
  int j=0;
  int k=0;
  if(arr1[size1-1]<arr2[size2-1]){ //decide if arr1 or arr2
      will write the last element in c
    while(j<size2){
      while(i<size1){
        if(arr1[i]<arr2[j]){
          c[k] = arr1[i];
          k++;
          i++;
        }
        else{
```

```
          c[k] = arr2[j];
          k++;
          j++;
        }
      }
      c[k]=arr2[j];
      k++;
      j++;
    }
  }
  else{
    while(i<size1){
      while(j<size2){
        if(arr1[i]<arr2[j]){
          c[k] = arr1[i];
          k++;
          i++;
        }
        else{
          c[k] = arr2[j];
          k++;
          j++;
        }
      }
      c[k]=arr1[i];
      k++;
      i++;
    }
  }
}

int partition(int* arr, int left, int right, int option){
    //partition for local quick sort.
  int i = left;
  int tmp;
  int pivot;
  if(option == 0){
    pivot = arr[right];
  }
  else if(option == 1){
    pivot = arr[(left+right)/2];
    arr[(left+right)/2] = arr[right];
    arr[right] = pivot;
  }
  else{
    pivot = arr[right];
  }
  //printf("i: %d \t, j: %d \t, pivot index: %d \t, pivot: %d
      \t",i,j,(left+right)/2,pivot);
  i = left - 1;
  for(int k = left; k < right; k++){
    if (arr[k] <= pivot){
      i++;
      tmp = arr[i];
      arr[i] = arr[k];
      arr[k] = tmp;
    }
  }
  tmp = arr[i+1]; //switch the pivot element to the correct
      place
  arr[i+1]=arr[right];
  arr[right] = tmp;
```

```
  return i+1; // return the index of pivot.
}

void quicksort(int* arr, int left, int right, int option){
    //local quick sort
  if (left < right){
    int pindex = partition(arr, left, right, option);
    if(left<pindex) { quicksort(arr,left, pindex-1,option);}
    if(pindex+1<right) { quicksort(arr, pindex+1,
        right,option);}
  }
}

void copyArray(int *d1, int *d2, int start, int length){
//Copy d1 elements from index start with length steps into
    vector d2
    int i;
    int j = start;
    for (i = 0; i < length; i++) {
      d2[i] = d1[j];
      j++;
    }
}


int read_file(char *name, int** pp){
//return the number of number read in the file and pointer *pp
    will point to the first element
    FILE* f;
    f = fopen(name, "r");

    if(f){
        fseek(f, 0, SEEK_END);
        fseek(f, 0, SEEK_SET);
        int *p = NULL;
        int n;
        fscanf(f,"%d ",&n);
        p = (int*)malloc(n*sizeof(int));
        for(int i=0;i<n;i++){
          fscanf(f,"%d ", &p[i]);
        }
        *pp = &p[0];
        fclose(f);
        return n;
    }
    else{
      printf("error with open your input file.\n");
      return 0;
    }
}

void save_result(char* name, int* arr, int n){
  FILE* f;
  f = fopen(name,"w");
  // fprintf(f, "%d ", n);
  for(int i=0; i<n; i++){
    fprintf(f,"%d ",arr[i]);
  }
  fclose(f);
}
```

```
int* mpi_qsort(int* data, int len, MPI_Comm com, int option){
  MPI_Status status;
  MPI_Request req;
  int size, rank;
  MPI_Comm_size(com, &size);
  MPI_Comm_rank(com, &rank);
  int pivot;
  int* data_neighbour;
  int num_neighbour=0;
  int len_lo,len_hi;
  int* data_lo;
  int* data_hi;
  int *mean_median = NULL;

  if(size == 1){
    MPI_Isend(data,len, MPI_INT, 0, 444, MPI_COMM_WORLD, &req);
    MPI_Request_free(&req);
    return data;
  }

  if(option == 1){ //strategy 1
    if(rank == 0){pivot = data[len/2];} //set pivot to the
        middle of processoe 0.
    MPI_Bcast(&pivot, 1, MPI_INT, 0, com);
  }
  else if(option == 2){ //strategy 2
    int processor_median = data[len/2];

    if (rank == 0) {
        mean_median = malloc(sizeof(int) * size);
    }
    MPI_Gather(&processor_median, 1, MPI_INT, mean_median, 1,
        MPI_INT, 0, com);
    if(rank == 0){
      quicksort(mean_median, 0, size, 1);
      pivot = mean_median[size/2];
      free(mean_median);
      mean_median = NULL;
    }
    MPI_Bcast(&pivot, 1, MPI_INT, 0, com);
  }

  else{ //strategy 3
    int processor_median = data[len/2];

    if (rank == 0) {
        mean_median = malloc(sizeof(int) * size);
    }
    MPI_Gather(&processor_median, 1, MPI_INT, mean_median, 1,
        MPI_INT, 0, com);
    if(rank ==0){
      long int median_average = 0;
      for (int k=0; k<size; k++){
        median_average = median_average + mean_median[k];
      }
      pivot = median_average/size;
      free(mean_median);
      mean_median = NULL;
    }
    MPI_Bcast(&pivot, 1, MPI_INT, 0, com);
  }

```

```c
int i = 0;
while(i<len && data[i]<pivot){i++;} //divide the local sorted
    array to small and large
len_lo = i;
len_hi = len-i;
data_lo = (int*)malloc(len_lo*sizeof(int));
data_hi = (int*)malloc(len_hi*sizeof(int));

for(int j=0;j<i;j++) {data_lo[j]=data[j];} //write data to
    the left part low
for(int j=i;j<len;j++) {data_hi[j-i]=data[j];} //write data
    to the right part high

//below to exchange data:
int len_new;
if(rank < size/2){
  MPI_Isend(data_hi,len_hi, MPI_INT, rank+size/2, rank, com,
      &req);
  MPI_Probe(rank+size/2, rank+size/2, com, &status);
  MPI_Get_count(&status, MPI_INT, &num_neighbour);
  data_neighbour = (int*)malloc(num_neighbour*sizeof(int));
  MPI_Recv(data_neighbour, num_neighbour, MPI_INT,
      rank+size/2,rank+size/2,com, MPI_STATUS_IGNORE);

  data = realloc(data, (len_lo+num_neighbour)*sizeof(int));
      //realloc memory to do merge after receiving the other
      part
  local_merge(len_lo, num_neighbour, data_lo, data_neighbour,
      data);
  len_new = len_lo + num_neighbour;
}

else{
  MPI_Probe(rank-size/2,rank-size/2,com, &status);
  MPI_Get_count(&status, MPI_INT, &num_neighbour);
  data_neighbour = (int*)malloc(num_neighbour*sizeof(int));
  MPI_Recv(data_neighbour, num_neighbour,
      MPI_INT,rank-size/2, rank-size/2, com,
      MPI_STATUS_IGNORE);
  MPI_Isend(data_lo, len_lo, MPI_INT, rank-size/2, rank,
      com,&req);

  data = realloc(data, (len_hi+num_neighbour)*sizeof(int));
  local_merge(len_hi, num_neighbour, data_hi, data_neighbour,
      data);
  len_new = len_hi + num_neighbour;
}

MPI_Wait(&req, &status);
free(data_lo);
free(data_hi);
free(data_neighbour); //data is sorted so we can free the
    other dynamic allocated memory

MPI_Comm sub;
int color = rank/(size/2);
MPI_Comm_split(com, color, rank, &sub);
int n_size;
MPI_Comm_size(sub, &n_size);
return mpi_qsort(data, len_new, sub, option);
```

```
}
```

quicksort.c