Uppsala University

Parallel and Distributed Programming

1TD070

Project - report

Authors:
Csongor Horváth

May 3, 2023

The problem setting

The problem was to implement the Parallel Quicksort algorithm using C and MPI. The algorithm (Algorithm 1) and the usage of the code, which includes the 3 given parameter, the necessary outputs and the format of the I/O files was described in the assignment, so I won't describe them here.

Algorithm 1 The Parallel Quick-sort algorithm

1. Divide the data into p equal parts, one per process
2. Sort the data locally for each process
3. Perform global sort
   - 3.1 Select pivot element within each process set
   - 3.2 Locally in each process, divide the data into two sets according to the pivot (smaller or larger)
   - 3.3 split the processes into two groups and exchange data pairwise between them so that all processes in one group get data less than the pivot and the others get data larger than the pivot.
   - 3.4 Merge the two sets of numbers in each process into one sorted list
4. Repeat 3.1 - 3.4 recursively for each half until each group consists of one single process.

This concludes the brief description of the problem setting. Information about my implementation and further detatils can be found in the next section.

Implementation

For the implementations it could be assumed that p, so the number of processes is a power of 2. Now there are I/O processes, which similarly to the previous assignments is done by the root process. I use rank 0 as root. As this processes are very similar to the previous assignment I won't explain them in details. I use functions for the read and write process and the run times are collected using `MPI_Reduce` to get the value to print in the screen in the root process.
I wrote two function to check if p is a 2 power and if it is then get $log_2(p)$, as these values needed in the project. Then I wrote a serial quicksort in 3 function which is needed in the 2nd step of the algorithm. And I implemented a merge function for the step 3.4 in the algorithm for testing. Also there is a print_list function which was used for testing.

Now let's talk about the assignment specific details. First note that even if the algorithm was given in a recursive form my implementation wasn't following a recursive way as knowing p the depth of the recursion is known to be $log_2(p)$, so instead of doing things recursively I used a loop and not like in the serial implementation I also split the pivot element into one of the sub processes, so in the end it was enough to gather the local list from each processor in the root to get the final ordered list. My hope was that in this way the list doesn't build up in a recursive manner, therefore it could be faster even without the left of the pivot elements, which wouldn't count for much considering the large size of the problem comparison to the dept of the recursion.

One nice assumption would be that $p \mid n$, so n dividable by p, but as it was asked so my program is working for arbitrary n value in the following manner: if $n < p$ then as the number of cores are relatively small, so in this case I only do a serial sort in the root process. If $p \mid n$, then I simply use `MPI_Scatter` to distribute the list between the processes. In other cases I make a calculation and divide as equally as possible (max difference is 1 between the processes in the list size). I use `MPI_Scatterv` for it after calculating the necessary parameters.

After distribution comes the 2nd step of the algorithm, so the local serial sort. I used quicksort for this, which may not be the best choice. As it will be explained later this is the most crucial part of the whole process, so it is recommended to choose a sorting algorithm here which is suitable for the use case of our application.

After this starts the recursive part, which is implemented in a loop as it's depth is given as I mention. The splitting of the groups are solved with the splitting of the communicator using `MPI_Comm_split` for splitting half each group from the previous state.

Now the next step is to get a pivot element, for this I use a function. It is a complicated process to get the pivot element as for this purpose the communication between the threads are required and also there are 3 case based on the user. Also the possibility that some of the processes has empty list in a given step makes this process more complicated as in this case the calculation of the pivot element is getting more complex with the consideration of which element is empty in the group.
The basic idea of the implementation is to gather all necessary data at one node in each groups and make the calculation locally there and finally distribute the pivot element in the group. Everything else is only the details for the lot of different cases and the simply calculation of the required values in each cases.

After the pivot element the splitting of the data comes as the following. First I calculate indexes, where the data smaller and larger than the pivot, and with which process should the informations be exchanged. Then I communicate the size of the exchange in both direction and then if the size is positive then I send and receive the data. I used `MPI_Isend` with `MPI_Wait` and `MPI_Recv` for communication here as the non-blocking send made the two way exchange more simple in code compared to the blocking, where I should pay attention to the order for the information exchange.

And finally I merge the list using a merge function. Note that the merge process is fast and simple as the two partial list are both ordered.

Now the above steps run in a loop $log_2(p)$ times instead of the recursion until there is one process in each group, but the result is the same this way.

In the end I gather the length of the distributed lists and then gather the partial list from the processes in a way to get a fully ordered list from them in the root process.

For the experiment note that in this case the time measurement includes the distribution and the gathering of the list.

Partitioning strategies

The following partitioning strategies are implemented in the project:

1. Select the median in one processor in each group of processors.
2. Select the median of all medians in each processor group.
3. Select the mean value of all medians in each processor group.

There are other possible pivot strategies, but as it will be explained in this algorithm it is not the most crucial part of the algorithm.

Performance experiment

First I would like to note that I didn't presented run times for the backward input files as in that case my choice of the serial sorting algorithm was pretty bad, since that is the worst case for the serial quick sort algorithm. Also note that even if for the given file the run time was too large, I made measurements for smaller backward files which was already slow with smaller number of integers.
Here I would like to mention that I made partial measurements and I find it that in most cases the majority of run times comes from the serial sorting. In the smaller (10^6) backwards case it was up to 95%. But even with the random large lists the serial sorting in the beginning mostly takes more than 50% of the run time. Therefore if we know something about our application it is crucial to choose a suitable sorting algorithm, or do something to avoid it's weaknesses. E.g: in this case if I would like to make it usable to the backward sorted lists, then I could shuffle the list elements before or during the distribution.
After discussing this I will start the discussion of my experiment which was made using the large input files (*input125000000.txt*, *input250000000.txt*, *input500000000.txt*, *input1000000000.txt*, *input2000000000.txt*).

pivot	-n 1	-n 2	-n 4	-n 8	-n 16	-n 32	-n 64	-n 128
1	14.8555	8.1279	4.6732	2.8762	1.6016	1.1451	2.5540	7.8167
2	14.8592	8.1570	4.6980	2.8975	1.6006	1.1614	3.0454	7.9721
3	14.8587	8.5494	5.7424	4.3978	3.4562	3.2196	4.6182	9.3997

Table 1: Time measurement for *input125000000.txt* in 2 node 32 core using pivot strategy 1,2,3

My first goal was to compare the 3 different pivot strategy. For this end I made measurements with all 3 pivot strategy with $2^i,\ i = 0, 1, \dots 7$ core. I used the smallest large input file. The run times can be seen in Table 1. For 64 and 128 core the measurements are not good as I only used 2 node (32 core) for measurements and these were run using `-oversubscribe` flag. But it is good to see that it is not worth to try using oversubscribe in this

application as there are projects, where it would further improve or at least not make the performance significantly worse as in this case.
And as for the pivot strategies the 1 and 2 seems to be very similar with 1 being a slightly, almost immeasurably faster, but for some reason 3 is significantly worse and the gap is getting bigger as using more core. Honestly I don't know why it is becoming worse, maybe it is slower to calculate, or it makes things worse in other way.
I want to note that again that the starting serial sort is the most time consuming in most time, as after that not considering the communication and the calculation of pivot I only merge sorted lists, qhich is a very fast thing to do even for large amount of number.

Strong scaling experiment

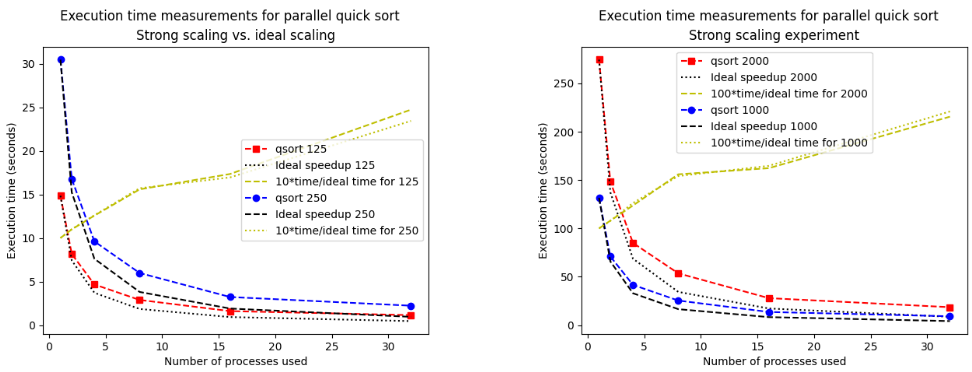
size 10^6	serial	-n 1	-n 2	-n 4	-n 8	-n 16	-n 32
125	14.5551	14.8526	8.1641	4.6673	2.8931	1.6104	1.1479
250	29.6503	30.4716	16.7777	9.6095	5.9846	3.2291	2.2321
500	61.6230	62.995683	34.8779	19.8823	12.3496	6.6020	4.4787
1000	132.3010	131.6470	70.9468	41.5660	25.3717	13.5538	9.0914
2000	280.9389	274.6057	148.3555	84.9485	53.5367	27.8613	18.5007

Table 2: Time measurement for strong scaling in 2 node 32 core using pivot strategy 1

For the strong scaling I made measurements with all of the large simple input files using $2^i,\ i = 0, 1, \ldots, 5$ cores. For reference I also made measurements with a simple serial implementation of the problem. As we can see the starting point in one core is not much worse then the specific serial implementation, it is even faster in some cases, which is probably a measurement error.

For better visualization I plotted the run times for most cases in below Figure 1 with the ideal speed up and the ratio between the ideal speed up and the real run time.
As it can be seen the strong scaling is relatively good, the ratio is similar as

I get in previous assignments and it is almost the same between the different files. The measurements is also similar in a way, that the ratio gets worse when reaching the limitation of our resources. My suggested reasons was explained for this in the previous assignment.



(a) quick sort run times

(b) quick sort run times

Figure 1: Strong scaling

Weak scaling experiment

core / size 10^6	$-n$ 1 / 125	$-n$ 2 / 250	$-n$ 4 / 500	$-n$ 8 / 1000	$-n$ 16 / 2000
quick sort	14.8526	16.7777	19.8823	25.3717	27.8613

Table 3: Time measurement for weak scaling in 2 node 32 core

For the weak scaling I used a subset of the run times measured above. The chosen run times can be seen in Table 3. Note that it is not easy to make a good weak scaling experiment as the scaling of the run time is not nicely describable. As in quick sort the worst case is $\mathcal{O}(n^2)$, but based on this we got a negative weak scaling as for this when changing input size to twice the size, then we should multiply the used number of cores with 4. A list of run times would be e.g:(125/1 : 14.8526, 250/4 : 9.6095 500/16 : 6.6020). So instead I used a linear scaling approach and I considered changing the file size to double as multiplying the used number of cores. Then I made a plot in Figure 2. Here fore ideal speed up I plotted the curve coming from the average run time $\mathcal{O}(nlog(n))$ and I used 1 as constant value in the $\mathcal{O}$ run time.

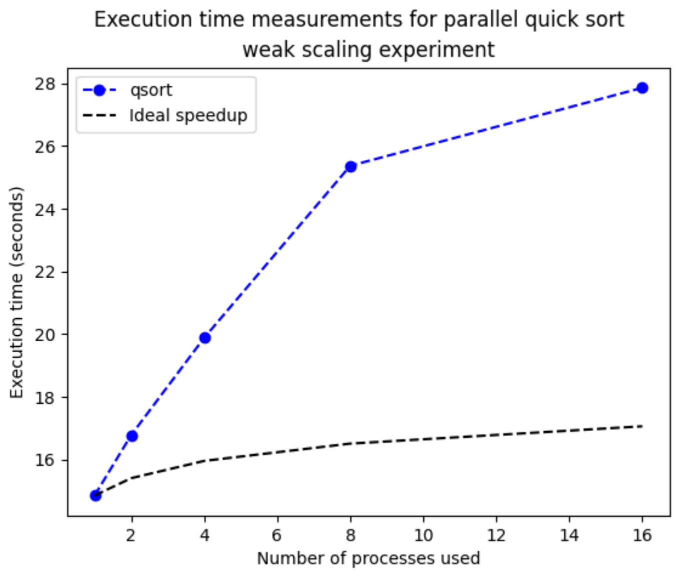


Figure 2: Weak scaling for quick sort

The weak scaling in this way looks quite bad, but I don't know for what I should measure the weak scaling I could use a squared scaling as that is the worst case, which would mean a line, but the real scaling is better then that. And I have no idea what constant I should use for the average case, so $\mathcal{O}(nlog(n))$ scaling. But I would say the tendency is a bit similar in the measured times with the $\mathcal{O}(nlog(n))$.

So in conclusion I would say that weak scaling experiment doesn't make much sense as we don't know what scaling we should expect from the algorithm for ideal case.

Summary

So in summary I would conclude that it can be useful to do the quick sort in a parallel manner as in experience I get a significant improvement until I get resource for it by using more core. And if in a really large scale application a sorting is needed, or in a relatively large scale it needed multiple times, then the parallel implementation can save us a lot of time.

I also would like to mention that a crucial part of this algorithm is the serial sort, so for real application we should we careful with it. Also the choice of the pivot can make impact on the run time.

There are also other possibilities to make a parallel implementation of quick sort. One example is to use the serial quick sort, but until there are empty processes let's start one of the recursive calls in new process.

Appendix

quicksort

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>


#define root 0
#define save 1

int read_input(const char *file_name, int **values) {
	FILE *file;
	if (NULL == (file = fopen(file_name, "r"))) {
		perror("Couldn't open input file");
		return -1;
	}
	int num_values;
	if (EOF == fscanf(file, "%d", &num_values)) {
		perror("Couldn't read element count from input file");
		return -1;
	}
	if (NULL == (*values = malloc(num_values * sizeof(int)))) {
		perror("Couldn't allocate memory for input");
		return -1;
	}
	for (int i=0; i<num_values; i++) {
		if (EOF == fscanf(file, "%d", &((*values)[i]))) {
			perror("Couldn't read elements from input file");
			return -1;
		}
	}
	if (0 != fclose(file)){
		perror("Warning: couldn't close input file");
	}
	return num_values;
}

int write_output(char *file_name, const int *output, int num_values) {
	FILE *file;
	if (NULL == (file = fopen(file_name, "w"))) {
		perror("Couldn't open output file");
		return -1;
	}
	for (int i = 0; i < num_values; i++) {
		if (0 > fprintf(file, "%d ", output[i])) {
			perror("Couldn't write to output file");
		}
	}
	if (0 > fprintf(file, "\n")) {
		perror("Couldn't write to output file");
	}
	if (0 != fclose(file)) {
		perror("Warning: couldn't close output file");
	}
	return 0;
}

int is_power_of_two(int x){
    return (x != 0) && ((x & (x - 1)) == 0);
}

int log_2(int x){
    int count=0;
    int pow=1;
    while(x > pow){
        count++;
        pow*=2;
    }

    return count;
}
```

```
// Codes for serial sorting
    int partition(int** arr, int low, int high) {
    int pivot = *(*arr + high);
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (*(*arr + j) < pivot) {
        i++;
        int temp = *(*arr + i);
        *(*arr + i) = *(*arr + j);
        *(*arr + j) = temp;
        }
    }
    int temp = *(*arr + i + 1);
    *(*arr + i + 1) = *(*arr + high);
    *(*arr + high) = temp;

    return (i + 1);
    }

    void quickSort(int** arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
    }

    void serial_sort(int** arr, int length) {
    quickSort(arr, 0, length - 1);
    }

int print_list(int* arr, int n){
    for (size_t i = 0; i < n; i++)
    {
        printf("%d ",arr[i]);
    }
    printf("\n");

}

int get_pivot(int median, int is_empty ,MPI_Comm* comm, int rank, int size, int pivot_strategy){
        int res; int all_empty; int sum;
        int* medians=(int*)malloc(size*sizeof(int));
        MPI_Allreduce(&is_empty, &all_empty, 1, MPI_INT, MPI_SUM, *comm);
        MPI_Gather(&median, 1, MPI_INT, medians, 1, MPI_INT, root, *comm);
        if(!all_empty){
                if(rank==root){
                switch (pivot_strategy)
                        {
                        case 1:
                                res=median;
                                break;

                        case 2:
                                serial_sort(&medians, size);
                                res=medians[size/2];
                                break;

                        case 3:
                                sum=0;
                                for (int i = 0; i < size; i++)
                                        sum +=medians[i];
                                res=sum/size;
                                break;
                        }

                for (int i = 0; i < size; i++)
                        medians[i]=res;
                }
        }
        else{ // case where some of the processes has currently empty sets
                int* empties=(int*)malloc(size*sizeof(int));
                MPI_Gather(&is_empty, 1, MPI_INT, empties, 1, MPI_INT, root, *comm);
                if(rank==root){
```

```
                        res=0;
                        switch (pivot_strategy)
                                {
                                case 1:
                                        for (int i = 0; i < size; i++){
                                        if(!empties[i]){
                                                res=medians[i];
                                                break;
                                        }
                                        }
                                        break;

                                case 2:
                                        int max=0;
                                        for (int i = 0; i < size; i++){
                                                if(medians[i]>max)
                                                        max=medians[i];
                                        }
                                        for (int i = 0; i < size; i++){
                                                if(empties[i]){
                                                        medians[i]=max+1;
                                                }
                                        }
                                        serial_sort(&medians, size);
                                        res=medians[(size-all_empty)/2];
                                        break;

                                case 3:
                                        int sum=0;
                                        for (int i = 0; i < size; i++){
                                                if(empties[i]){
                                                        sum += medians[i];
                                                }
                                        }
                                        if(size-all_empty!=0)
                                        res=sum/(size-all_empty);
                                        break;
                                }
                for (int i = 0; i < size; i++)
                        medians[i]=res;
                }
                free(empties);
        }
        // scattering pivot value
        MPI_Scatter(medians, 1, MPI_INT, &res, 1, MPI_INT, root, *comm);
        free(medians);
        return res;
}

void merge_lists(int* lis1, int* lis2, int len1, int len2, int** arr_pointer){
        int* merge;
        if(len1+len2==0){
                merge=NULL;
        }
        else{
        int i=0; int j=0;
        merge = (int*)malloc((len1+len2)*sizeof(int));
        while (i<len1 && j<len2)
        {
                if(lis1[i]<lis2[j]){
                        merge[i+j]=lis1[i];
                        i++;
                }
                else{
                        merge[i+j]=lis2[j];
                        j++;
                }
        }
        while(i<len1){
                merge[i+j]=lis1[i];
                i++;
        }
        while(j<len2){
                merge[i+j]=lis2[j];
                j++;
        }
        }
```

```c
        free(*arr_pointer);
        *arr_pointer = merge;
}

int main(int argc, char **argv) {
        if (4 != argc) {
                printf("Usage: qsort input_file output_file pivot_strategy(1-3)\n");
                return 1;
        }
        char *input_name = argv[1];
        char *output_name = argv[2];
        int pivot_strategy = atoi(argv[3]);


        int *local_data=NULL;
    int *input=NULL;
        int* output=NULL;

    double run_time;
    double max_runtime;
    int size; int log2size;
    int rank;
    int n; int length; int length_shift=0;
    int median, pivot;
        int goal_idx, send_size, cnt, rec_size;
        int idx_send, idx_use;
        int* buffer=NULL;
        int* sendc=NULL; int* recc=NULL; int* displ=NULL;
        int i;
        int is_empty=0;



        MPI_Request send_req1;
        MPI_Request send_req2;


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

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);


if(!is_power_of_two(size))//checking if number of processor really is a power of 2
    {printf("The number of processors shouldd be a powwer of 2!\n"); return 0;}

log2size = log_2(size);
MPI_Comm comm[log2size];
int ranks[log2size];
int sizes[log2size];
int color;

//Setting up the data
    if(rank==root){
    // Read input file
        if (0 > (n = read_input(input_name, &input))) {
                return 2;
        }
    }


        // Start timer
        MPI_Barrier(MPI_COMM_WORLD);
        double start = MPI_Wtime();


//ALG step 1 - Distributing data as equal as possible
    MPI_Bcast(&n, 1, MPI_INT, root, MPI_COMM_WORLD);
    length=n/size;

if(n<size){
        if(rank==root){
                double start = MPI_Wtime();
                serial_sort(&input,n);
                run_time=MPI_Wtime()-start;
                output=input;
```

```
        }
}
else{
    if(n%size==0){ // Distribution if it is possible equally
            local_data=(int*)malloc(length*sizeof(int));
            MPI_Scatter(input, length, MPI_INT, local_data, length, MPI_INT, root,
                MPI_COMM_WORLD);
        }

    else{ // Distribution if it is impossible equally
        length++;
        local_data=(int*)malloc(length*sizeof(int));
        if(rank >= n%size)
            length_shift=-1;

        sendc=(int*)malloc(size*sizeof(int));
        displ=(int*)malloc(size*sizeof(int));
        for (int i = 0; i < size; i++)
        {
            if (i < n%size)
                {sendc[i]=length;
                if(i!=0)
                    displ[i]=displ[i-1]+length;
                else
                    displ[i]=0;}
            else
                {sendc[i]=length-1;
                                if (i==n%size)
                                        displ[i]=displ[i-1]+length;
                                else
                        displ[i]=displ[i-1]+(length-1);}
        }
        MPI_Scatterv(input, sendc, displ, MPI_INT, local_data, length, MPI_INT, root,
            MPI_COMM_WORLD);
                length=length+length_shift;
                free(sendc);free(displ);
        }

        if(rank==root)
                free(input);

// ALG 2 - Sort the data locally for each process
        serial_sort(&local_data, length);

//ALG 3 - Perform global sort (Note even if it a recursive algorithms it is implemented
    withouth recursion as the depth is fix)
        MPI_Comm_dup(MPI_COMM_WORLD,&(comm[0]));
        for (i = 0; i < log2size; i++)
        {
        // Setting up the splitting of the communication
                MPI_Comm_size(comm[i], &(sizes[i]));
        MPI_Comm_rank(comm[i], &(ranks[i]));
                color=ranks[i]/(sizes[i]/2);
                MPI_Comm_split(comm[i], color, ranks[i], &(comm[i+1]));

        //ALG 3.1 get pivot element with
                if(length==0){
                        is_empty=1;
                        median=0;}
                else{
                        is_empty=0;
                        median=local_data[(length)/2];}
                pivot=get_pivot(median, is_empty, &comm[i], ranks[i], sizes[i],
                    pivot_strategy);

        //ALG 3.2 divide data according to pivot
                //finding splitting points
                for (cnt = 0; cnt < length; cnt++){
                        if(local_data[cnt]>pivot){
                                break;
                        }
                }

                //setting up indexes to exchange data
                if(ranks[i] < sizes[i]/2){
                        goal_idx=ranks[i]+sizes[i]/2;
                        send_size=length-cnt;
```

```
                        idx_send=cnt;
                        idx_use=0;
                }
                else{
                        goal_idx=ranks[i]-sizes[i]/2;
                        send_size=cnt;
                        idx_send=0;
                        idx_use=cnt;
                }

        //ALG 3.3 Communicate data in the given pair
                //Note: it is possible that some list is empty in this process, which needs some care
                MPI_Isend(&send_size, 1, MPI_INT, goal_idx, 0, comm[i],&send_req1);
                MPI_Recv(&rec_size, 1, MPI_INT, goal_idx, 0, comm[i], MPI_STATUS_IGNORE);
                MPI_Wait(&send_req1, MPI_STATUS_IGNORE);

                if (send_size!=0)
                        MPI_Isend(&(local_data[idx_send]), send_size, MPI_INT, goal_idx, 0, comm[i], &send_req2);

                if(rec_size!=0){
                        buffer=(int*)malloc(rec_size*sizeof(int));
                        MPI_Recv(buffer, rec_size, MPI_INT, goal_idx, 0, comm[i], MPI_STATUS_IGNORE);
                }

                if (send_size!=0)
                        MPI_Wait(&send_req2, MPI_STATUS_IGNORE);

        //ALG 3.4 merge the local and recived list - modify local length
                merge_lists(&(local_data[idx_use]), buffer, length-send_size, rec_size, &local_data);
                length=length-send_size+rec_size;
                free(buffer);
                buffer=NULL;
        }

//ALG 5 - Gather sorted data in the root
        recc=(int*)calloc(size,sizeof(int));
        displ=(int*)malloc(size*sizeof(int));

        //Get the length of the arrays in the processes
        MPI_Gather(&length, 1, MPI_INT, recc, 1, MPI_INT, root, MPI_COMM_WORLD);
        displ[0]=0;
        for (int i = 1; i < size; i++){
                        displ[i]=displ[i-1]+recc[i-1];
        }

        //Gather the final ordered list
        if(rank==root)
                output=(int*)malloc(n*sizeof(int));
        MPI_Gatherv(local_data, length, MPI_INT, output, recc, displ, MPI_INT, root, MPI_COMM_WORLD);

//Stop clock

        run_time = MPI_Wtime()-start;
        //get the measured runtime
        MPI_Reduce(&run_time, &max_runtime, 1, MPI_DOUBLE, MPI_MAX, root, MPI_COMM_WORLD);

        // Clean up
        free(local_data);
    free(recc);
        free(displ);
}

        // Write results from root
    if(rank==root){
                printf("%lf\n", max_runtime);

                #if save
                if (0 != write_output(output_name, output, n)) {
                        return 2;
                }
```

```
                free(output);
                #endif
    }

        if(n>=size)
        for (int i = 0; i < log2size; i++){
                MPI_Comm_free(&comm[i]);
        }

        MPI_Finalize();
        return 0;
}
```

Makefile

```
################################################################################
# Makefile for assignment 2, Parallel and Distributed Computing 2023.
################################################################################

CC = mpicc
CFLAGS = -std=c99 -g -O3
LIBS = -lm

BIN = quicksort

all: $(BIN)

quicksort: quicksort.c
	$(CC) $(CFLAGS) -o $@ $< $(LIBS)

clean:
	$(RM) $(BIN)
```