Mini Project Report

ON

Evaluate performance enhancement of parallel Quicksort Algorithm using MPI

BY

Akshay Kumar Mishra

Roll No.: 27

Under the Guidance

of

Prof. Poonam S. Jadhavar

Department of Computer Engineering

Padmabhooshan Vasantdada Patil Institute of Technology

Bavdhan, Pune-21

Savitribai Phule Pune University

[2022 - 23]

\

Department of Computer Engineering
Padmabhooshan Vasantdada Patil, Institute of Technology,
Bavdhan, Pune

CERTIFICATE

This is to certify that **Akshay Kumar Mishra** from **Final Year Computer Engineering** has successfully completed his mini project titled **Evaluate performance enhancement of parallel Quicksort Algorithm using MPI at** Padmabhooshan Vasantdada Patil Institute of Technology, Bavdhan in the partial fulfillment of the Bach- elors Degree in Engineering.

Prof. Poonam S. Jadhavar	Prof. Ganesh Wayal
Mini Project Guide	H.O.D

\

Acknowledgement

We take this opportunity to thank our guide and Head of the Department **Prof. Ganesh Wayal** for their valuable guidance and for providing all the necessary facilities, which were indispensable in the completion of this project report. We are also thankful to all the staff members of the Depart- ment of Computer Engineering for their valuable time, support, comments, suggestions and persuasion. We would also like to thank the institute for providing the required facilities, Internet access and important books.

Akshay Kumar Mishra
Roll No. 27

Contents:

Problem statement:

Quicksort Algorithm Implementation:

Quicksort is a classic example of a Divide and Conquer family of algorithms. In this technique, to solve a problem we "break" it into subproblems, solve each subproblem separately and combine the individual solutions to produce the final solution of the problem. Another name for this sort is "partition exchange sort". As the first name of the algorithm indicates, it is a very efficient classification method. As its second name indicates, its main features are twofold: exchange and partitioning. First, swaps are performed between array elements so that elements with smaller values are moved to one side, while elements with larger values are moved to the other side of the array. Thus, the application of partitioning the table into two smaller ones that are sorted independently of each other follows. It is obvious that the two sub-arrays are treated the same way: exchange and partition.

Theoretical presentation

Quicksort is a sorting algorithm developed by the British Tony Hoare. The algorithm is based on the "divide andconquer" technique. At each step of the algorithm:

- A random element (pivot) A[q] of the array A is selected • Splits: the elements of A are divided into those smaller than the pivot and those which are greater than the pivot.
- Ruled: The same process is recursively applied to each of A's segmentsthat came up

Quick Sort Pseudocode:

```
quickSort(ÿ, p, r) {
    if (p < r) {
        q = Partition(A, p, r);
        quickSort(A, p, q-1);
        quickSort(A, q+1, r);
    }
}
```

Pivot selection and partitioning can be done in many different ways. The choice of specific implementationplans significantly affects the performance of the algorithm.

Lomuto Partition:

This figure is attributed to Nico Lomuto. The last element of the table is selected as the pivot. This algorithm maintains a pointer i as it scans the array using another pointer j so that the elements in low to i-1 are less thanthe pivot and the elements in i to j are equal to or greater than the pivot. Since this scheme is more compact and understandable, it is often used for introductory teaching
material, although it is less effective than Hoare's original scheme e.g. In the event that all elements are thesame. In the worst case when the array is already sorted the complexity is O(n^2)

Lomuto Partition Pseudocode:

```
1. partition_lomuto(A, low, high)
       2.pivot := A[high] i :=
       low
3.
4.     for j := low to high do if A[j] < pivot
5.             then swap A[i] with A[j]
6.
7.                     i := i + 1
8.      swap A[i] with A[high] 9. return i
```

Hoare Partition:

The original scheme described by Tony Hoare uses two pointers that start at the ends of the partitioned array, then move back and forth until they find a reversal: a pair of elements, one greater than or equal to the pivot, and one less or equal to the pivot, which are in the wrong order with each other. The elements are then exchanged. When the pointers meet, the algorithm stops and returns the final pointer. Hoare's scheme is moreefficient than Lomuto's partition system because it makes three times fewer swaps on average and creates efficient
partitions even when all values are equal.

Lomuto Partition Pseudocode:

```
1. partition_hoare(A, low, high) pivot := A[ÿ(high
         + low) / 2ÿ] 2.
3.       i := low - 1
4.                 j := high + 1
loop
5.                 forever
6.                           do
                                       i := i + 1
7.
8.                             while A[i] < pivot
9.                           do
10                                     j := j - 1
.                            whileA[j] > pivot
11
.        if i ÿ j
         then                          return j
12.
13
.
14.                swap A[i] with A[j]
```

Complexity

- **Worst case:** The worst behavior for quicksort occurs when the partitioning routine produces one subproblem/subarray with n-1 elements and one with 0 elements. Assume that an unbalanced partitionresults in each recursive call. The partition cost is O(n). The callback on an array of size 0 returns:

(0) = (1)

The recursion for the running time is: ( ) = (ÿ 1)
+ (0) + ( ) = (ÿ 1) + ( ) If we sum the cost incurred at
each level of the recursion, we get a numerical sequence, which evaluates the complexity as ( ).
2

- **Ideal Case:** The ideal behavior for quick sort occurs when the partitioning routine produces two subproblems/subarrays each of size no larger than n/2, since one is of size [n/2] and one is of size[(n/ 2)-1]. The runtime regression is:

( ) = 2 ( ) + ( ) 2

This recursion results in complexity ( ).

• **Average/Expected case:**

We can get an idea of the average case by considering a case where the partition puts O(n/9)elements into one set and O(9n/10) elements into another set. Here is a replay for this case.

9 ( ) = ( ) + (9
10) + ( )

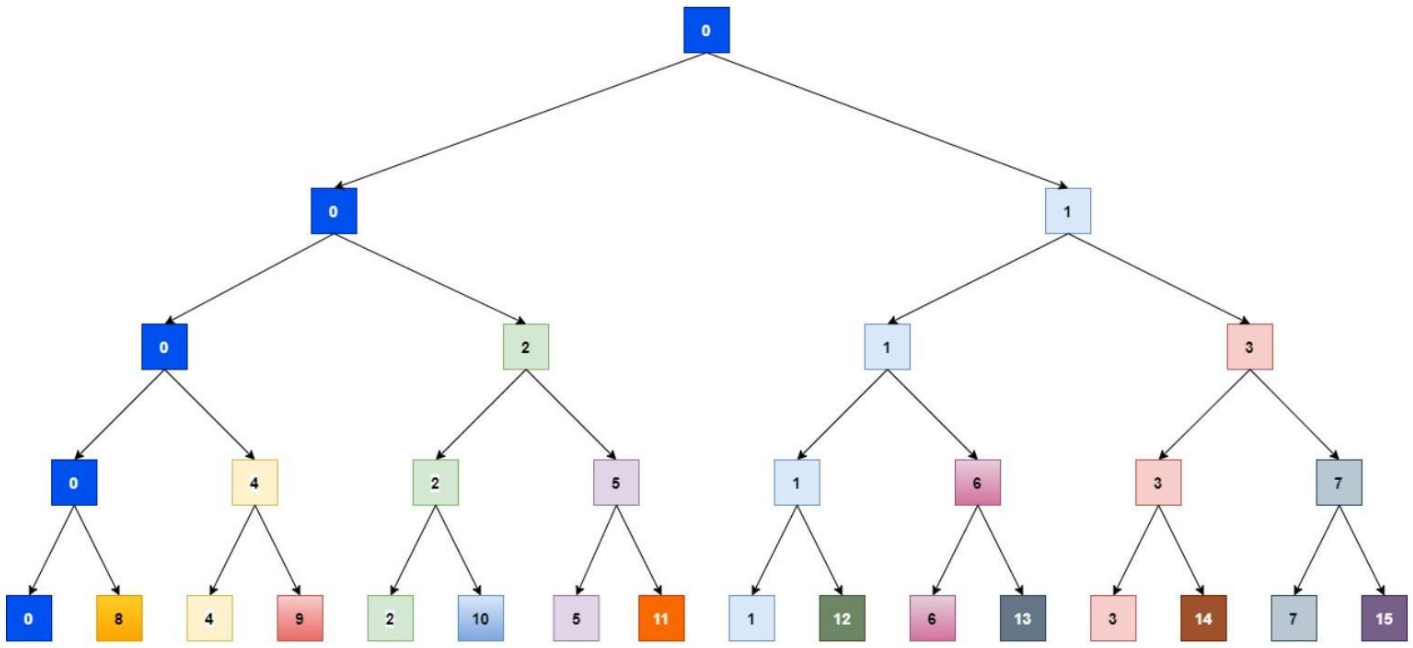
The solution to the above iteration is also complexity ( ), where the notation ( ) hides aconstant slightly larger than the ideal case.

Parallel Approach

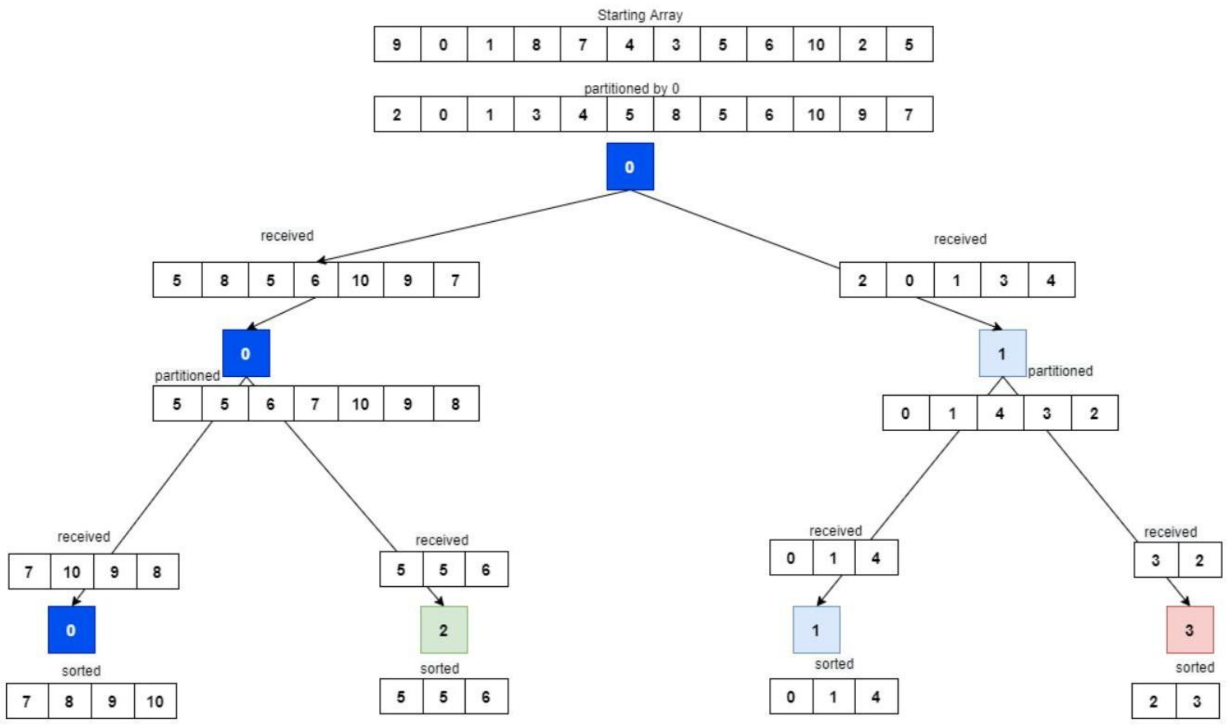
In this work, the parallel version of quicksort was implemented with two methods, the recursiveand the merge.

Quicksort Recursive - Recursive Quicksort

In this version, we follow the philosophy of "divide and conquer". The main computer (Master) isthe one who will start the process and distribute a piece of the table to another computer and keep the rest. This process is repeated for all computers until no one is available to accept a subarray. At this point, all computers sort their piece with the serial algorithm and return the result they received back to the computer that sent it. The diagram below explains the process:

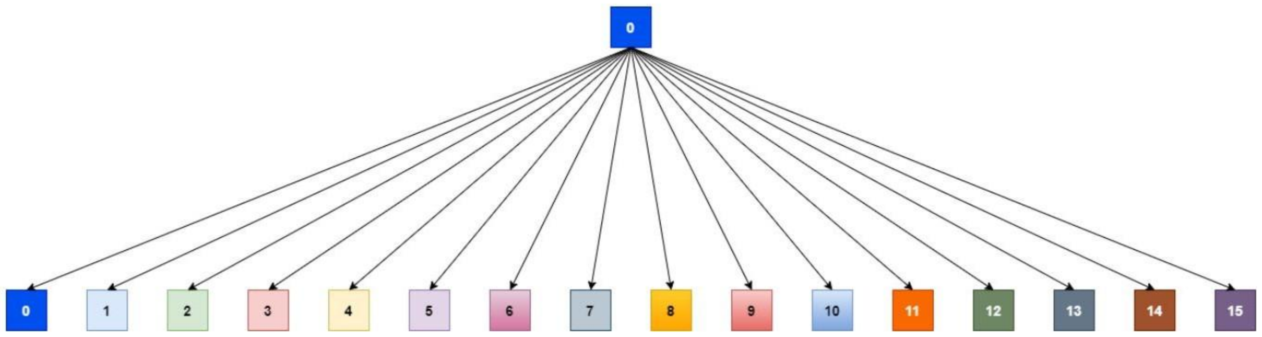

0
0
1
0
2
1
3
0
4
2
5
1
6
3
7
0
8
4
9
2
10
5
11
1
12
6
13
3
14
7
15

For a better understanding of the process flow, the following example is given:



Merge Quicksort - Quicksort with union

In the Merge Quicksort implementation we divide the original array into sub-arrays which we sendto the respective computers. After each computer has sorted its table, it sends it to the original computer where the tables are successively joined to form the final, now sorted, table.



Performance and Metrics

During the implementation of the parallel algorithm, all the necessary metrics were measured against the maximum capabilities of the platform in order to determine its performance. Therefore, for Cluster Size = 16, three different dimensions of the problem are studied, fortable sizes of 100,000, 500,000 and 1,000,000 positions.

Recursive Implementation

Time Complexity

The serial execution of the algorithm with a data array of 10000000 elements requires time **Ts=2.383101 seconds** The parallel executionof the algorithm with a data array of
10000000 elements from 16 computers requires time **Tp=1.317556 seconds**

Speedup - Speedup

The speedup we get by running the algorithm in parallel with a data array of 10000000 elements from 16 computers:
2.383101 =1.808728434 1.1317556

= =

Efficiency - Efficiency

The efficiency we get by running the algorithm in parallel with a data array of 10000000 elements from 16 computers:1.808728434 = 0.1130455271

= =

16

Problem Size - Size

Our serial algorithm, Quicksort with Lomuto Partitioning, in the average/general case has complexity (So for a 1000000 element dataarray the problem size will be).

Communication Cost - Communication Cost

By taking measurements, we measure the time required for a round trip communicationbetween two computers to send an array of size x. The linear curve that best fits and describes the data is as follows:

ÿ7 = 1.6668538 ÿ + 0.0007423201
10

Therefore, wanting to measure the communication cost for running the algorithm with 16 computers (ideally), we consult the above diagram where the following results. At the 1stlevel of the co~~mmunicati~~on tree, **a** table exchange of size about N/2 takes place. At the 2nd lev~~el of the c~~ommunication tree, **two** matrix exchanges of size about N/4 take place. At the 3rd level of the

co~~mmunicati~~on tree, **four** matrix exchanges of size about N/8 take place. At the 4th level ofthe
communication tree there are **eight** table exchanges of size approximately N/16.

Therefore for the table of 100,000 seats the communication
cost is: = 0.009 + 2 ÿ 0.0049 + 4 ÿ 0.0028 + 8
ÿ 0.00178 = 0.044

For the table of 500,000 places the communication cost is: =
0.0424 + 2 ÿ 0.0215 + 4 ÿ 0.0111 + 8 ÿ 0.0059 = 0.177 For the table of 1,000,000 places the communication cost is: =
0.084 + 2 ÿ0.0424 + 4 ÿ 0.0215 + 8
ÿ 0.0111 = 0.3436

Therefore, the values for the communication cost of the specific implementationare described by the following curve:

= 3.329016 ÿ 10
ÿ + 0.01065246
7

Merge Implementation

Time Complexity

The serial execution of the algorithm with a data array of 10000000 elements requires time
Ts=2.752389 seconds The
parallel execution of the algorithm with a data array of 10000000 elements from
16 computersrequires time **Tp=0.397279 seconds**

Speedup - Speedup

The speedup we get by running the algorithm in parallel with a data array of 10000000elements from 16 computers: 2.752389 =

= = 6.928100906

0.397279

Efficiency - Efficiency

The efficiency we get by running the algorithm in parallel with a data table of10000000 elements from 16 computers:
6.928100906

= 0.4330063067

= =

16

Problem Size - Size

Our serial algorithm, Quicksort with Hoare Partitioning, in the average/generalcase has complexity ( ). So for a data array of 1000000 elements the problem size will be 10 6 10 6 .

Communication Cost - Communication Cost

Having calculated a linear curve that fits and best describes the data is the following:

= 1.6668538 ÿ 10 ÿ7 + 0.0007423201

In the case of Merge, the diagram shows that we have **15** exchanges of tables of size N/16.(15 because the first piece is held by computer 0). So for 16 computers
and a board of size 100,000 seats, the cost is:

= 15 ÿ 0.00178 = 0.0267s

So for 16 computers and a board of size 500,000 seats, the cost is:

$= 15 ÿ 0.00595 = 0.08925s$

So for 16 computers and a board of size 1,000,000 seats, the cost is:

$= 15 ÿ 0.0111 = 0.1665s$

Therefore, the values for the communication cost of the specific implementationare described by the following curve:

= 1.552992 ÿ 10

ÿ + 0.01132377

7

Problem Complexity Analysis

With the general assumption of time complexity for sequential quicksort= n log n, Let's take
the ideal resolution for the above application, assuming that each partition can
create two equalpartitions. For two processors:

( ) = 2 ( ) 2

For three processors:

( ) = (– (–) – (–)) <=>
+

2 2 4

( ) = 2

4

For four processors:

( ) 2

( ) = 4

( ) 4

Hence, **for p number of processors:**

ÿ1 < < 2 +1

( ) = ( ), where
2

But since the application works with shared smallest partition and variable partition
size, this timeanalysis cannot prove the average execution time for parallel
quicksort(Quicksort Recursive).

Also, the experimental results also proved that this time complexity does not hold for the averageexecution of the application.

In the case of the 2nd method, Parallel Quicksort with Merge, the distribution of the arrays doesnot need to be done in powers of 2, but is distributed among all available computers. That is, thetime complexity of this method depends on the size of the subarray that all computers will receive. So each computer will run the serial quicksort on its subarray. For **an n-seat array andp available computers,** the time complexity is:

$$(\ ) = \qquad (\ )$$

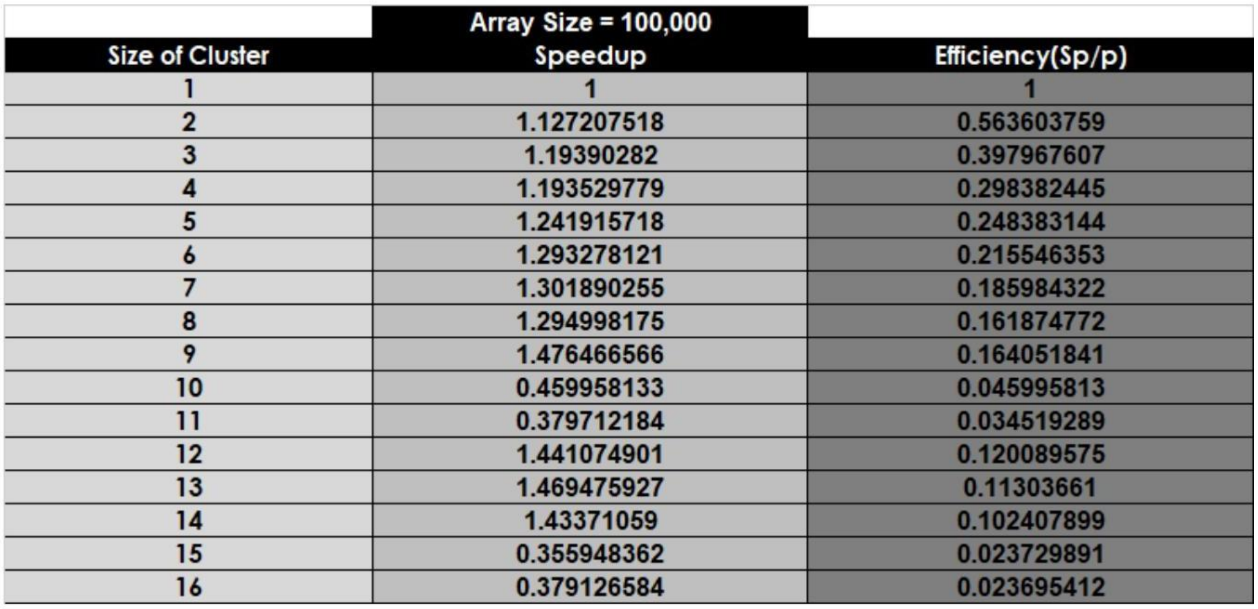
Of course, this is not a representative size as a large part of the calculation is also taken up injoining all the sub-tables back to the final table.

As the tables are joined the 1st with the 2nd, the 3rd with the union of the previous two and soon. This process is performed (p-1) times, where the average array size is (n/2 + n/p) and thus:

( ) = − +(2 — +)(ÿ 1) ( )

Experimental results

Recursive Implementation

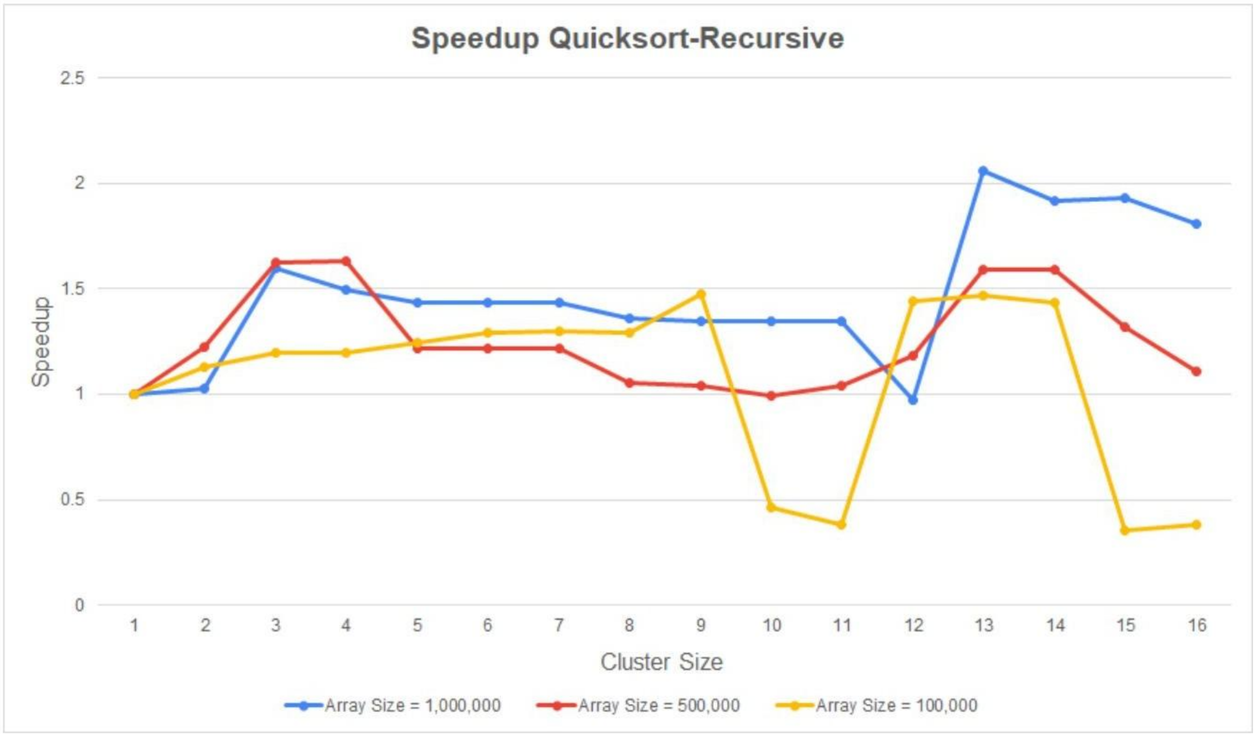


	Array Size = 100,000	
Size of Cluster	Speedup	Efficiency(Sp/p)
1	1	1
2	1.127207518	0.563603759
3	1.19390282	0.397967607
4	1.193529779	0.298382445
5	1.241915718	0.248383144
6	1.293278121	0.215546353
7	1.301890255	0.185984322
8	1.294998175	0.161874772
9	1.476466566	0.164051841
10	0.459958133	0.045995813
11	0.379712184	0.034519289
12	1.441074901	0.120089575
13	1.469475927	0.11303661
14	1.43371059	0.102407899
15	0.355948362	0.023729891
16	0.379126584	0.023695412

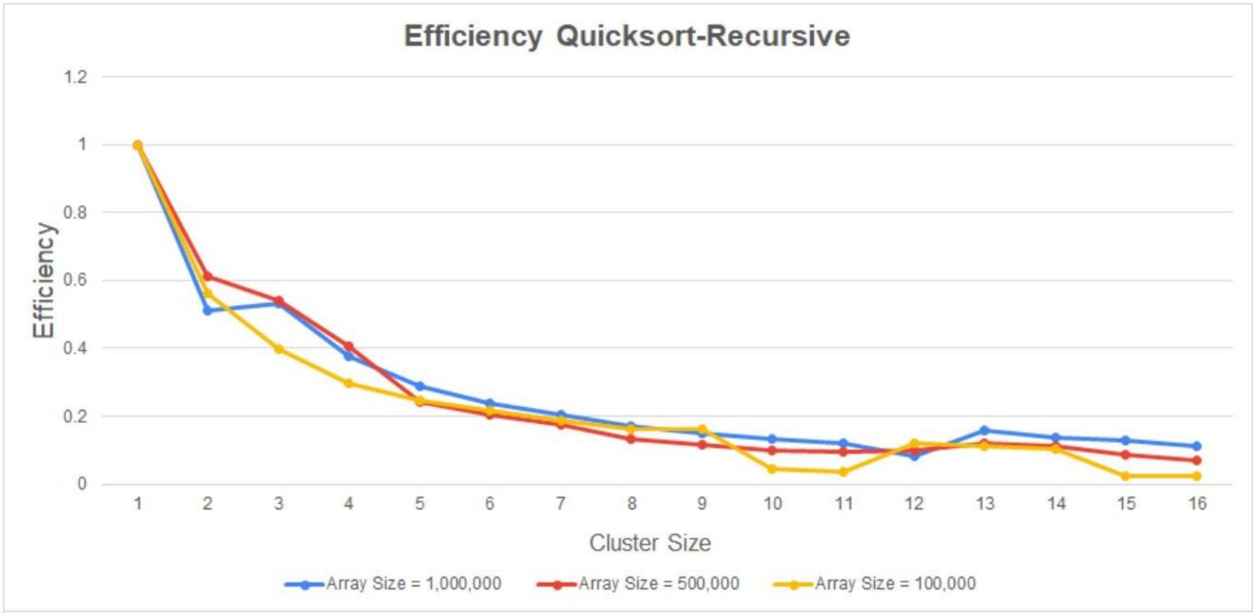
	Array Size = 500,000	
Size of Cluster	**Speedup**	**Efficiency(Sp/p)**
1	1.00000	1
2	1.227394784	0.613697392
3	1.626107888	0.542035963
4	1.62901	0.407253303
5	1.214000268	0.242800054
6	1.217235312	0.202872552
7	1.21551	0.173644198
8	1.053031161	0.131628895
9	1.043787784	0.11597642
10	0.98992	0.098991644
11	1.042411073	0.094764643
12	1.182560938	0.098546745
13	1.59008	0.122313769
14	1.589139287	0.113509949
15	1.319996779	0.087999785
16	1.11066	0.069416355

	Array Size = 1,000,000	
Size of Cluster	**Speedup**	**Efficiency (Sp/p)**
1	1	1
2	1.02633517	0.513167585
3	1.59962961	0.53320987
4	1.499136602	0.37478415
5	1.437593126	0.287518625
6	1.435759562	0.23929326
7	1.433059763	0.204722823
8	1.359093421	0.169886678
9	1.344569198	0.149396578
10	1.348793523	0.134879352
11	1.345199147	0.122290832
12	0.973787689	0.081148974
13	2.059217047	0.158401311
14	1.919835173	0.137131084
15	1.929705188	0.128647013
16	1.808728434	0.113045527

Speedup(Sp=Ts/ Tp)



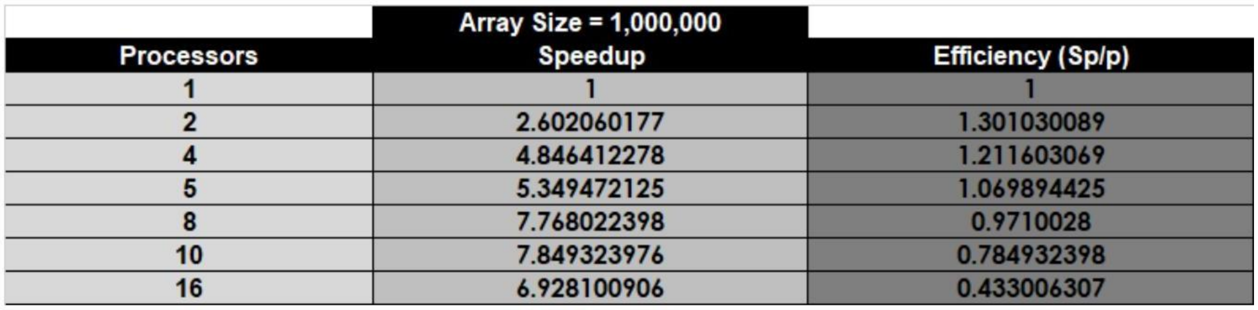
Efficiency (E=Sp/ p)



Merge Implementation

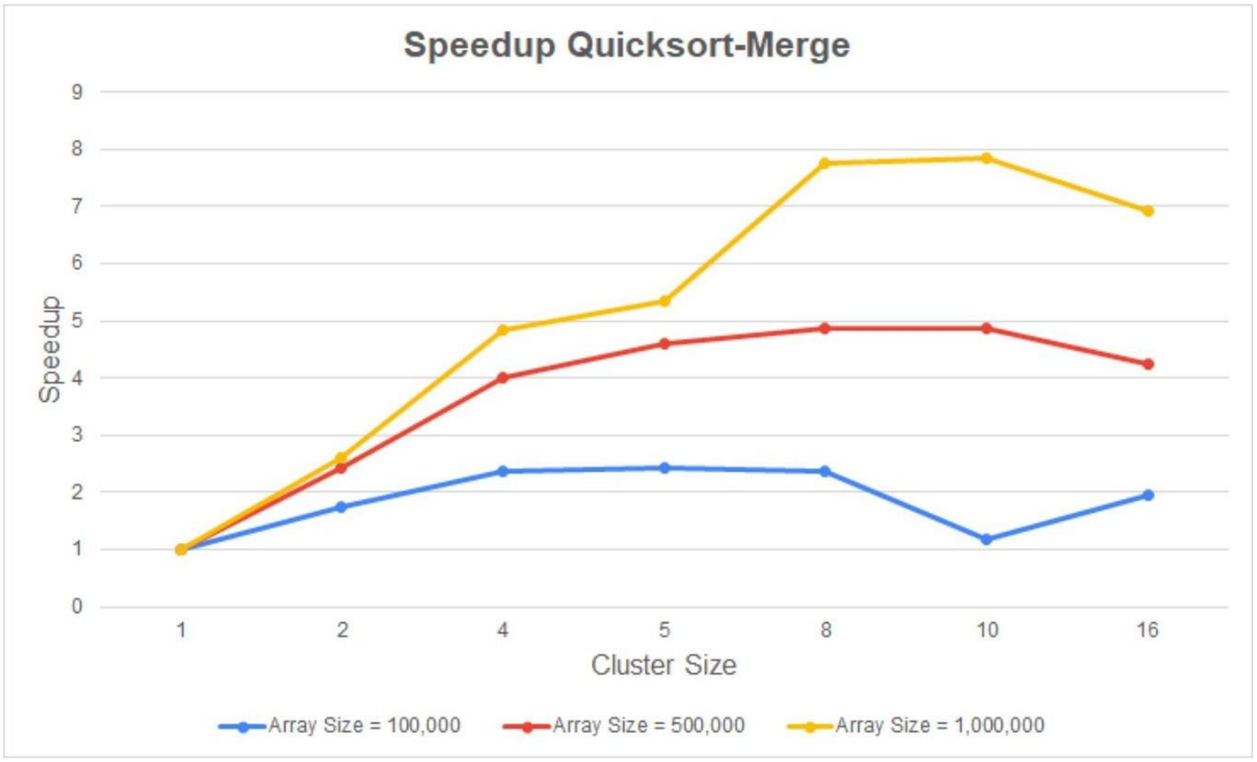
	Array Size = 100,000	
Processors	**Speedup**	**Efficiency (Sp/p)**
1	1	1
2	1.733937485	0.866968743
4	2.353924905	0.588481226
5	2.434428202	0.48688564
8	2.371999648	0.296499956
10	1.169716152	0.116971615
16	1.939213149	0.121200822

	Array Size = 500,000	
Processors	**Speedup**	**Efficiency (Sp/p)**
1	1	1
2	2.432054183	1.216027091
4	3.998815323	0.999703831
5	4.607969159	0.921593832
8	4.874180524	0.609272566
10	4.863757925	0.486375793
16	4.246359309	0.265397457

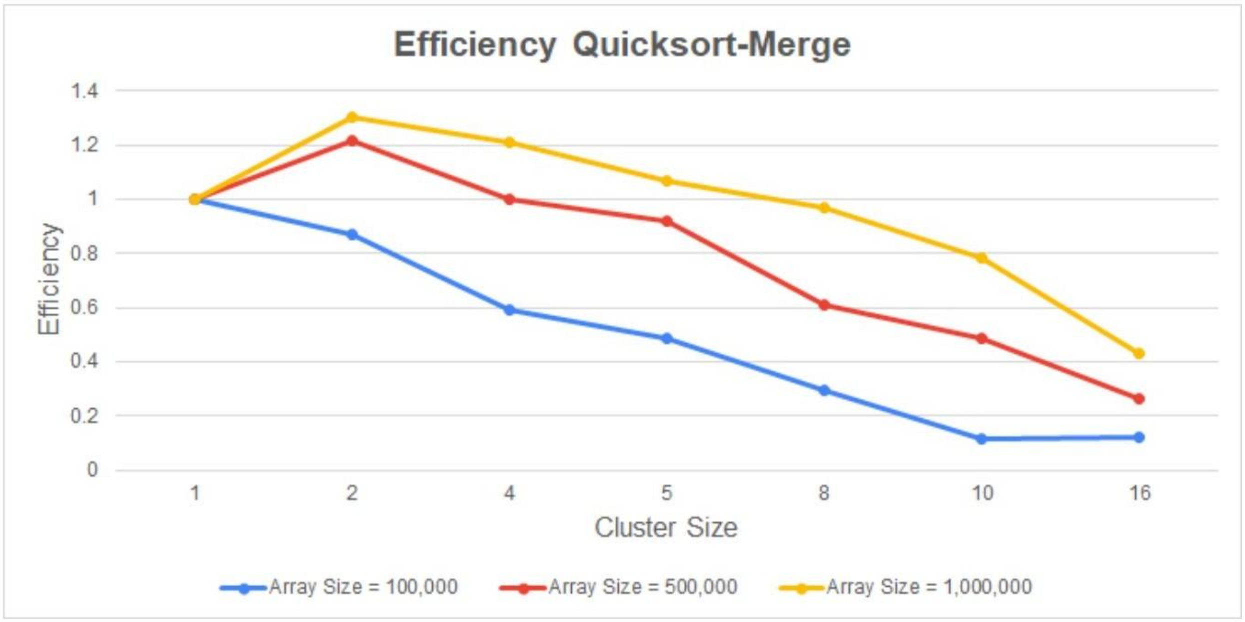


	Array Size = 1,000,000	
Processors	Speedup	Efficiency (Sp/p)
1	1	1
2	2.602060177	1.301030089
4	4.846412278	1.211603069
5	5.349472125	1.069894425
8	7.768022398	0.9710028
10	7.849323976	0.784932398
16	6.928100906	0.433006307

Speedup(Sp=Ts/ Tp)

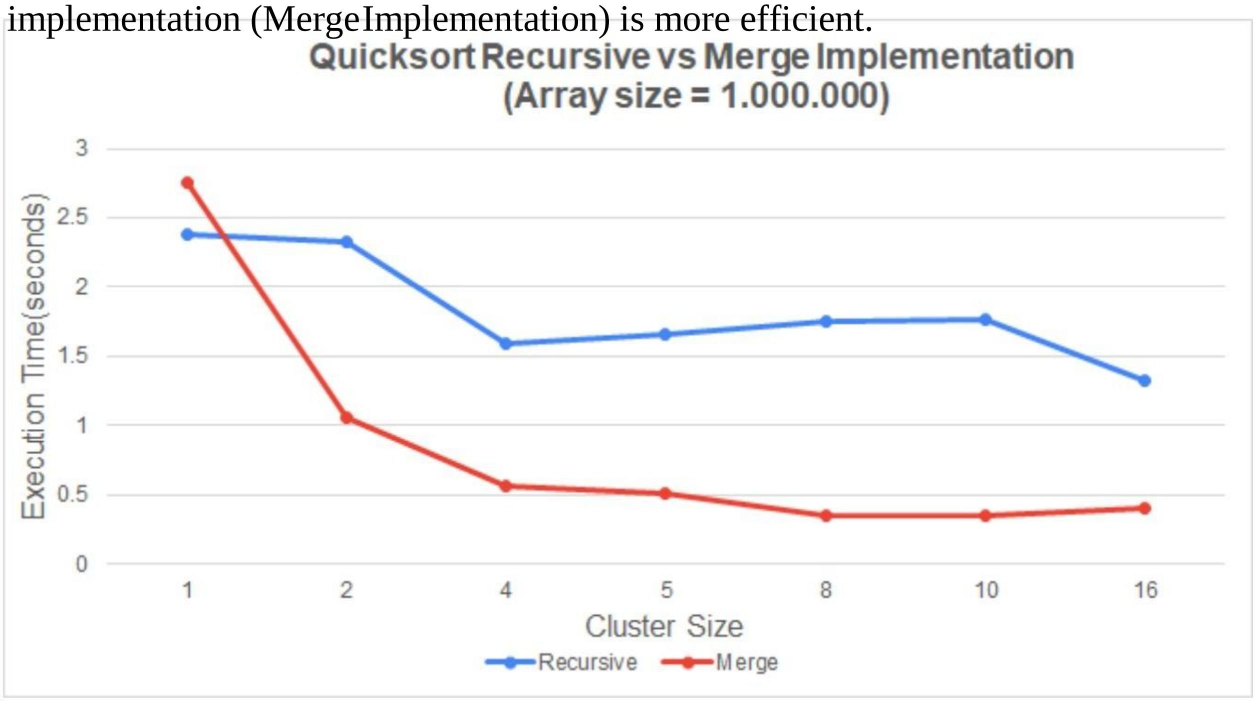


Efficiency (E=Sp/ p)



Conclusions

The diagram below shows the execution times of the two algorithms depending on thenumber of computers participating in the cluster, for data tables of size 1,000,000 elements. We notice that the hybrid implementation (MergeImplementation) is more efficient.



As the data shared by the algorithms is used by at most one processor at each stage, memorycoherency problems do not occur and timing control is not necessary to protect it. Therefore, shared memory platforms such as Openmp are able to support implementations of our algorithms without any change in their flow, significantlyreducing communication costs.

Quicksort Recursive implementation (C languageMPI platform)

```
#include
"mpi.h"
#include<st
dio.h>
#include
<stdlib.h>
#include
"math.h"
#include
<stdbool.h>
#define SIZE
1000000

/*
    Divides the array given into two partitions
        - Lower than pivot
        - Higher than pivot and returns
    the Pivot index in the array
*/
int partition(int *arr, int low, int
    high){ int pivot = arr[high]; int i
    = (low - 1); int j,temp; for
    (j=low;j<=high-1;j++)
    { if(arr[j] <
    pivot){ i++;
    temp=arr[i];
    arr[i]=arr[j];
    arr[j]=temp;




    } }
    temp=arr[i
    +1];
    arr[i+1]=a
    rr[high];
    arr[high]=t
    emp;
    return
    (i+1);
}

/*
    Hoare Partition - Starting pivot
    is the middle point Divides the
    array given into two partitions
        - Lower than pivot
        - Higher than pivot
    and returns the Pivot index in the array
*/
int hoare_partition(int *arr, int
    low, int high){ int middle =
    floor((low+high)/2); int pivot =
    arr[middle];
    int j,temp; // move
    pivot to the end
    temp=arr[mi
    ddle];
    arr[middle]=
    arr[high];
    arr[high]=te
    mp;

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

            arr[i]=
            arr[j];
            arr[j]=t
            emp;

        }
    } // move
    pivot back
    temp=arr[i+
    1];
    arr[i+1]=arr
    [high];
    arr[high]=te
    mp;

    return (i+1);
}

/*
    Simple sequential Quicksort Algorithm
*/
void quicksort(int *number,int
    first,int last){ if(first<last){ int
    pivot_index =
        partition(number, first, last);
        quicksort(number,first,pivot_index-1);
        quicksort(number,pivot_index+1,last);

    }
}

/*
    Functions that handles the sharing of subarrays to the right clusters
*/
int quicksort_recursive(int* arr, int arrSize, int
    currProcRank, int maxRank, int rankIndex) {
    MPI_Status status;

    // Calculate the rank of the Cluster which I'll send the other half int shareProc =
    currProcRank
    + pow(2, rankIndex);
    // Move to lower layer in the tree rankIndex++;


    // If no Cluster is available, sort sequentially by yourself and return if (shareProc >
    maxRank) {
        MPI_Barrier(MPI_COMM_
        WORLD); quicksort(arr, 0,
        arrSize-1 ); return 0;

    }
    // Divide array in two parts with the pivot in
    between int j = 0; int pivotIndex; pivotIndex =
    hoare_partit
    ion(arr, j,
    arrSize-1 );

    // Send partition based on size(always send the smaller part),
    // Sort the remaining partitions,
    // Receive sorted
    partition if (pivotIndex
    <= arrSize -
    pivotIndex) {
        MPI_Send(arr, pivotIndex , MPI_INT, shareProc, pivotIndex,
        MPI_COMM_WORLD); quicksort_recursive((arr + pivotIndex+1),
        (arrSize - pivotIndex-1 ), currProcRank,
maxRank, rankIndex);
        MPI_Recv(arr, pivotIndex , MPI_INT, shareProc, MPI_ANY_TAG,
        MPI_COMM_WORLD, &status);

    } else {
        MPI_Send((arr + pivotIndex+1), arrSize - pivotIndex-1, MPI_INT, shareProc,
        pivotIndex
+ 1, MPI_COMM_WORLD);
        quicksort_recursive(arr, (pivotIndex),
        currProcRank, maxRank, rankIndex);
        MPI_Recv((arr + pivotIndex+1), arrSize -
        pivotIndex-1, MPI_INT, shareProc,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    }
}
```

```
int main(int argc,
    char *argv[]) {
    int
    unsorted_array[S
    IZE]; int
    array_size =
    SIZE; int size,
    rank;
    // Start Parallel
    Execution
    MPI_Init(&arg
    c, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,  &rank);
    MPI_Comm_size(MPI_COMM_
    WORLD, &size);if(rank==0){
        // --- RANDOM ARRAY GENERATION ---
        printf("Creating Random List of %d
        elements\n", SIZE); int j = 0; for (j = 0; j <
        SIZE; +
        +j) { unsorted_array[j]
            =(int) rand() %
            1000;

        }  printf("Created\n");
    }

    // Calculate in which layer of the tree each
    Cluster belongs int rankPower = 0; while
    (pow(2, rankPower)
    <= rank){ rankPower++;

    }
    // Wait for all clusters to
    reach this point
    MPI_Barrier(MPI_COM
    M_WORLD); double
    start_timer, finish_timer;
    if (rank == 0) { start_timer
    =
        MPI_Wtime(); // Cluster
        Zero(Master) starts the Execution and // always runs recursively
        and keeps the left bigger half quicksort_recursive(unsorted_array, array_size,
        rank, size - 1, rankPower);  }else{  // All other Clusters wait for their subarray
        to arrive, // they sort it
    and
       they send it back.

        MPI_Status
        status; int
        subarray_siz
        e;
        MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG,
        MPI_COMM_WORLD, &status); //Capturing size of the
        array to receive
        MPI_Get_count(&status, MPI_INT,
        &subarray_size); int source_process =
        status.MPI_SOURCE; int
        subarray[subarray_size];
        MPI_Recv(subarray, subarray_size, MPI_INT,
MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        quicksort_recursive(subarray, subarray_size, rank, size - 1,
        rankPower); MPI_Send(subarray, subarray_size, MPI_INT,
        source_process, 0, MPI_COMM_WORLD);
    };

    if(rank==0)
        { finish_timer = MPI_Wtime();
        printf("Total time for %d Clusters : %2.2f sec \n",size, finish_timer-
        start_timer);

        // --- VALIDATION CHECK ---
        printf("Checking.. \n"); bool error
        = false; int i=0;

        for(i=0;i<SIZE-1;i++) {
            if (unsorted_array[i] >
                unsorted_array[i+1]){
                error = true;
                printf("error in i=%d \n", i);
            }
        }
```

```
            if(error)
                  printf("Error..Not sorted correctly\n"); else printf("Correct!
            \n");


      }

      MPI_Finalize();
      // End of Parallel Execution return 0;


}
```

Quicksort Merge implementation (C language platformMPI)

```
#include
"mpi.h"
#include<st
dio.h>
#include
<stdlib.h>
#include
"math.h"
#include
<stdbool.h>
#define SIZE
1000000

/*
     Hoare Partition - Starting pivot
     is the middle point Divides the
     array given into two partitions
          - Lower than pivot
          - Higher than pivot and returns
     the Pivot index in the array
*/
int hoare_partition(int *arr, int
     low, int high){ int middle =
     floor((low+high)/2); int pivot =
     arr[middle];
     int j,temp; // move
     pivot to the end
     temp=arr[mi
     ddle];
     arr[middle]=
     arr[high];
     arr[high]=te
     mp;

     int i = (low - 1);
     for
```

```
    (j=low;j<=high-
    1;j++){ if(arr[j] <
        pivot){ i++; temp=arr[i];

            arr[i]=
            arr[j];
            arr[j]=t
            emp;

        }

    } // move
    pivot back
    temp=arr[i+
    1];
    arr[i+1]=arr
    [high];
    arr[high]=te
    mp;

    return (i+1);
}

/*
    Simple sequential Quicksort Algorithm
*/
void quicksort(int *number,int
    first,int last){ if(first<last){ int
    pivot_index =
        hoare_partition(number, first, last);
        quicksort(number,first,pivot_index-1);
        quicksort(number,pivot_index+1,last);

    }
}

/*
    Function that handles the merging of two sorted subarrays
    and returns one bigger sorted array
*/
void merge(int *first,int *second, int
    *result,int first_size,int second_size){ int
    i=0; int
    j=0;
    int
    k=0
    ;

    while(i<first_size && j<second_size){

        if (first[i]<second[j])
            { result[k]=first[i]; k++; i+
            +; }


        else{ result[k]=second[j]; k++; j+
            +;


        }

        if(i == first_size){ // if the
            first array has been sorted while(j<second_size)
            { result[k]=second[j]; k++; j+
                +;


            }
        } else if (j == second_size){ // if the
            second array has been sorted while(i < first_size)
            { result[k]=first[i]; i++; k++;



            }
        }
    }
}

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

    int *unsorted_array = (int *)malloc(SIZE * sizeof(int)); int *result = (int
    *)malloc(SIZE * sizeof(int)); int
    array_size = SIZE; int size, rank; int
```

```
sub_array_size;


MPI_Status  status;
// Start parallel
execution
MPI_Init(&arg
c, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,
&rank);
MPI_Comm_size(MPI_COMM_WORLD,
&size);

if(rank==0){
    // --- RANDOM ARRAY GENERATION ---
    printf("Creating Random List of %d
    elements\n", SIZE); int j = 0; for (j =0; j <
    SIZE; +
    +j) { unsorted_array[j]
        =(int) rand() %
        1000;

    } printf("Created\n");
}
```

```
    // Number of computers in cluster int iter_count
    = size;
    // Determine the size of the subarray each computer
    receives sub_array_size=(int)SIZE/ iter_count;

    // Computer 0 (Master) splits the array and sends each subarray to the
    respective machine if( rank == 0 ){ double start_timer;
    start_timer=MPI_Wtime();
        int i =0; if(iter_count > 1){


            // ==========================================SENDING
DATA==========================================
            for(i=0;i<iter_count-
                1;i++){ int j; //send
                the
                subarray
MPI_Send(&unsorted_array[(i+1)*sub_array_size],sub_array_size,MPI_INT,i+1,0,MPI
_COMM_WORLD);
            }

            //
========================================C
ALCULATE FIRST
SUBARRAY================================
=========
            int i =0; int
            *sub_array = (int *)malloc(sub_array_size*sizeof(int));
            for(i=0;i<sub_array_size;i+
            +){
                // Passing the first sub array since rank 0 always calculates the first sub
array
                sub_array[i]=unsorted_array[i];
            }
            // Sequentially sorting the
            first array
            quicksort(sub_array,0,sub
            _array_size-1);

            // ==========================================RECEIVING
```

```
DATA==========================================
            for (i=0;i<iter_count;i++){ if(i > 0){  int

                temp_sub_array[sub_array_size];
                    // Receive each subarray

MPI_Recv(temp_sub_array,sub_array_size,MPI_INT,i,777,MPI_COMM_WORLD,&sta
tus); int j; int

                      temp_result[i*sub
                            _array_size];
                    for(j=0;j<i*sub_arr
                            ay_size;j++)
            { temp_result[j]=result[j];

                    } int temp_result_size = sub_array_size*i;
                    // Merge it back into the result array
                    merge(temp_sub_array,temp_result,result,sub_a
                    rray_size,temp_result_size);

                }else{
                    // On first iteration we just pass the sorted elements to the result array
                    int j; for(j=0;j<sub_array_size;j+
                    +)
                    {  result[j]=sub_array[j];


                    } free(sub_array);
                }

        } }
          else{ // if it runs only in a
            single computer
            quicksort(unsorted_arr
            ay,0,SIZE-1);
```

```
                        for(i=0;i<SIZE;i++)
                                { result[i]=unsorted_array[i];
                        }

                } double finish_timer;
                finish_timer=MPI_Wtime();
                printf("End Result: \n");
                printf("Cluster Size %d, execution time measured : %2.7f sec \n",size, finish_timer  start_timer); }else{ // All
the other

        computers have to sort the data and send it back sub_array_size=(int)SIZE/iter_count; int
                *sub_array = (int
                *)malloc(sub_array_size*sizeof(int));
                MPI_Recv(sub_array,sub_array_size,MPI_INT,0,0,MPI_COMM_WORLD,&status);
                quicksort(sub_array,0,sub_array_size-1); int i=0;

                MPI_Send(sub_array,sub_array_size,MPI_INT,0,777,MPI_COMM_WORLD);//sends the data back
to rank 0
                free(sub_array);
        }

        if(rank==0){ //
                --- VALIDATION CHECK ---
                printf("Checking.. \n"); bool
                error = false; int i=0;

                for(i=0;i<SIZE-1;i++) { if
                        (result[i] > result[i+1]){ error = true;
                                printf("error in
                                i=%d \n", i);
                        }

                } if(error)
                        printf("Error..Not sorted correctly\n"); else

                printf("Correct!\n");

        } free(unsorted_array);
        // End of Parallel Execution
        MPI_Finalize();
}
```

Bibliographical references

1. Pavlos Ephraimidis. Course Lectures Algorithms and Data Structures. Democritus University of Thrace. https://eclass.duth.gr/courses/TMA566/ 2.
Wikipedia contributors. Quicksort. Wikipedia. https://en.wikipedia.org/wiki/Quicksort 3. Cormen, TH (2009). Introduction to Algorithms, 3rd Edition (The MIT Press) (3rd ed.). MIT Press.
4. Foster, J. (1995). Designing and Building Parallel Programs. Addison-Wesley.
5. Miller, R., & Boxer, L. (2012). Algorithms Sequential & Parallel: A Unified Approach (3rded.). Cengage Learning.
6. Michael P. Bekakos. High Performance Computing Course Lectures: Parallel Algorithms and Computational Complexity. Democritus University of Thrace.https://eclass.duth.gr/courses/TMA503/