# CITS5507 – Project 1

*Chenxin Hu (22961779)*
*17/9/2021*

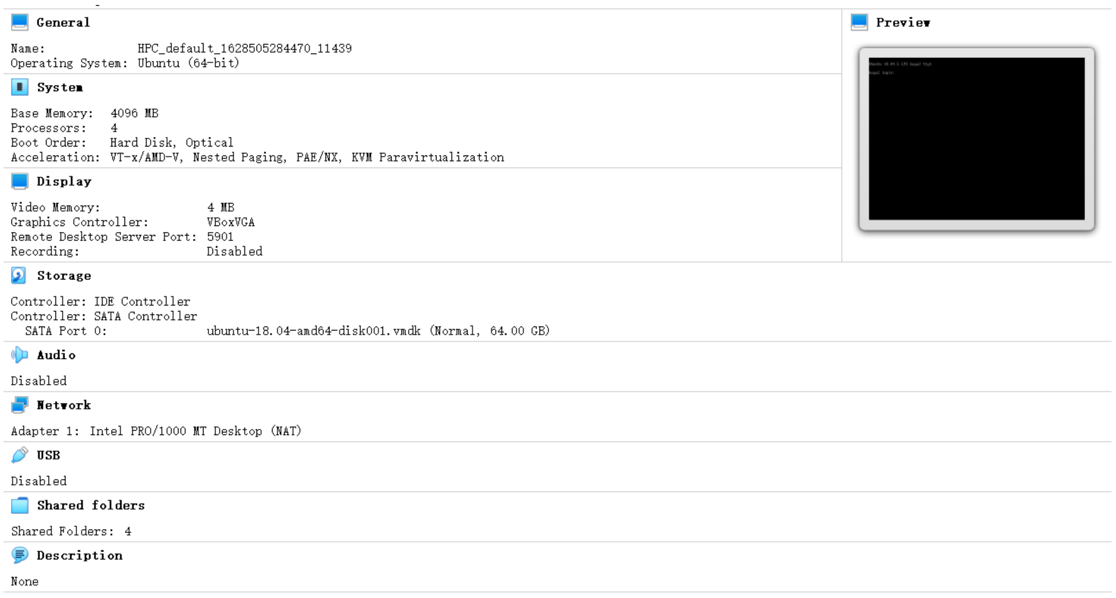
## Part1 Experiment design

### Overview

In this experiment, we develop the serial and parallel solutions for 3 sorting algorithms: quick sort, enumeration sort and merge sort. The solutions for each algorithms are tested on virtual machine, using VirutalBox and Vagrant, the result of time consumption for each solution are recorded and analyzed.

### Experiment Environment

The experiment is carried out on a virtual machine, the environment is shown below (Generated from VirtualBox):



```
General
Name:             HPC_default_1628505284470_11439
Operating System: Ubuntu (64-bit)

System
Base Memory:   4096 MB
Processors:    4
Boot Order:    Hard Disk, Optical
Acceleration:  VT-x/AMD-V, Nested Paging, PAE/NX, KVM Paravirtualization

Display
Video Memory:              4 MB
Graphics Controller:       VBoxVGA
Remote Desktop Server Port: 5901
Recording:                 Disabled

Storage
Controller: IDE Controller
Controller: SATA Controller
  SATA Port 0:             ubuntu-18.04-amd64-disk001.vmdk (Normal, 64.00 GB)

Audio
Disabled

Network
Adapter 1: Intel PRO/1000 MT Desktop (NAT)

USB
Disabled

Shared folders
Shared Folders: 4

Description
None

Preview
```

### Compile Codes

In this experiment, we use the
*mpic++ -fopenmp fileName.cpp*
instruction to compile the source code.

Using the instruction to run the executable file (a.out):

*mpirun -n 1 a.out*

Because we use multiple threads instead of multiple process, we set the parameter to 1.

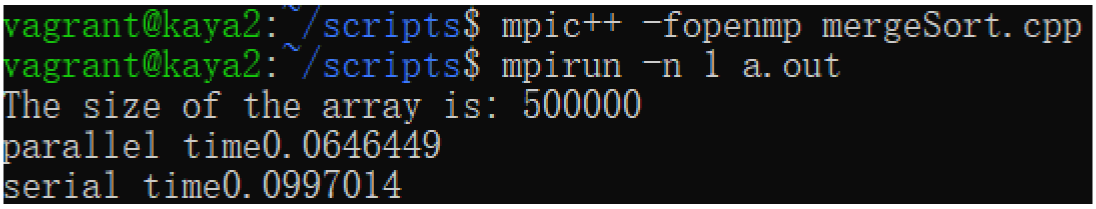
Also, using instruction:

*G++ -fopenmp filename.cpp -o executableFileName*

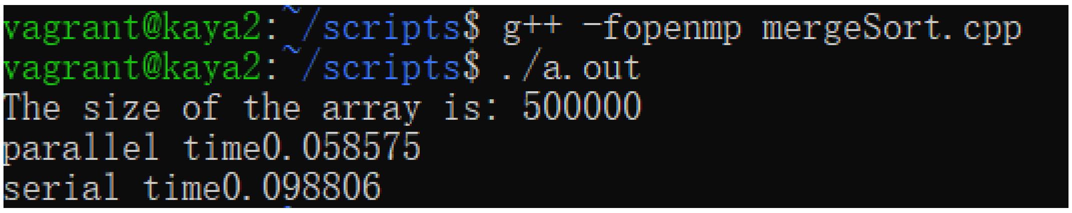
*./executableFileName.out*

Could work too.

Sample:



```
vagrant@kaya2:~/scripts$ mpic++ -fopenmp mergeSort.cpp
vagrant@kaya2:~/scripts$ mpirun -n 1 a.out
The size of the array is: 500000
parallel time0.0646449
serial time0.0997014
```



```
vagrant@kaya2:~/scripts$ g++ -fopenmp mergeSort.cpp
vagrant@kaya2:~/scripts$ ./a.out
The size of the array is: 500000
parallel time0.058575
serial time0.098806
```

## Pseudocodes

### Enumeration Sort Serial Solution

```
Generate a random array
Create a new empty array as temp array
For each item in the array
    while array not ending
        comparing the item with every item in the array.
        record how many items is smaller or equal to this item
    Add the item to the temp array in position number corresponding to record-1.
    #Cause the item Itself takes is one of these items that "smaller or equal to the item"#
For each item in the temp array
    If the item is empty #or, not fulfilled in the previous procedure
        Fulfill it with the nearest filled value which locates after the item.
        #Cause for one value shows many times in the array, the previous procedure only
records it once in the temp array, so between the two fulfilled value, there is an empty zone.
Fortunately, the empty value stands for the items have the same value but covered by the
latest shown one, so the empty value should equal to the nearest fulfilled right value. #
Copy the temp array to the original array.
Delete temp array
```

## Enumeration Sort Parallel Solution

```
Generate a random array
Create a new empty array as temp array
Using omp parallel to automatically divide the array to equal parts,
Each do the following procedure:
For each item in the array
    while array not ending
        comparing the item with every item in the array.
        record how many items is smaller or equal to this item
    Add the item to the temp array in position number corresponding to record-1.
Done parallel
    #Cause the item Itself takes is one of these items that "smaller or equal to the item"#
For each item in the temp array
    If the item is empty # or, not fulfilled in the previous procedure
        Fulfill it with the nearest filled value which locates after the item.
        #Cause for one value shows many times in the array, the previous procedure only records it once in the temp array, so between the two fulfilled value, there is an empty zone. Fortunately, the empty value stands for the items have the same value but covered by the latest shown one, so the empty value should equal to the nearest fulfilled right value. #
Copy the temp array to the original array.
Delete temp array
```

## Quick Sort Serial Solution

```
Generate a random array
Create two pointer points to edge of the array
If two pointers meet or go through each other, stop the function and return
Take the value of the item that left pointer points as sample value
Create two new pointers to record the current pointer
While two new pointers do not meet each other
    While new right pointer hasn't met the value larger or equal than the sample value
        Cursing left
    While new left pointer hasn't met the value smaller or equal than the sample value
        Cursing right
    Swap the items' value that two new pointers point.
Create new procedure, using the original left pointer as left pointer and new right pointer as right pointer
Create new procedure, using the original right pointer as right pointer and new right pointer as right pointer
```

**Quick Sort Parallel Solution**

```
Generate a random array
Create two pointer points to edge of the array
If two pointers meet or go through each other, stop the function and return
Take the value of the item that left pointer points as sample value
Create two new pointers to record the current pointer
While two new pointers do not meet each other
    While new right pointer hasn’t met the value larger or equal than the sample value
        Cursing left
    While new left pointer hasn’t met the value smaller or equal than the sample value
        Cursing right
    Swap the items’ value that two new pointers point.
If flag>0
    Set flag=flag-1
    Use two threads to do the following procedure separately:
    Create new procedure, using the original left pointer as left pointer and new right pointer as right pointer
    Create new procedure, using the original right pointer as right pointer and new right pointer as right pointer
    Done Parallel:
Else
Create new procedure, using the original left pointer as left pointer and new right pointer as right pointer
Create new procedure, using the original right pointer as right pointer and new right pointer as right pointer
```

**Merge Sort Serial Solution**

```
Generate a random array
Create two pointer points to edge of the array
If two pointers meet or go through each other, stop the function and return
Break the array from the middle into two parts, apply the same procedure, using two threads to do the two parts individually
Merge two parts
```

**Merge Procedure**

```
Create a temp array
```

```
While array1 not end and array2 not end
    If the item value in array1 smaller than array2
        Put the value into the temp array
        Right cursing array1
    Else
        Put the value into the temp array
        Right cursing array2
Put the remaining value in array1 or array2 to temp array
Delete temp array.
```

**Merge Sort Parallel Solution**

```
Generate a random array
Create two pointer points to edge of the array
If two pointers meet or go through each other, stop the function and return
If flag>0
    Set flag=flag-1
    Use two threads to do the following procedure separately:
    Break the array from the middle into two parts, apply the same procedure.
    Done Parallel:
Else
    Break the array from the middle into two parts, apply the same procedure.
Merge two parts
```

**Merge Procedure**

```
Create a temp array
While array1 not end and array2 not end
    If the item value in array1 smaller than array2
        Put the value into the temp array
        Right cursing array1
    Else
        Put the value into the temp array
        Right cursing array2
Put the remaining value in array1 or array2 to temp array
Delete temp array.
```

## Part2 Experiment Result and Analysis

### Overview

In this experiment, we tested the sorting speed for each algorithm's parallel solution and serial solution, using array size 10^3, 10^4, 10^5 and 5*10^5. Larger array size should work, but the virtual machine execution time when the array size gets greater than 5*10^5 become unacceptable, hence we'll not using the array size larger than 5*10^5.

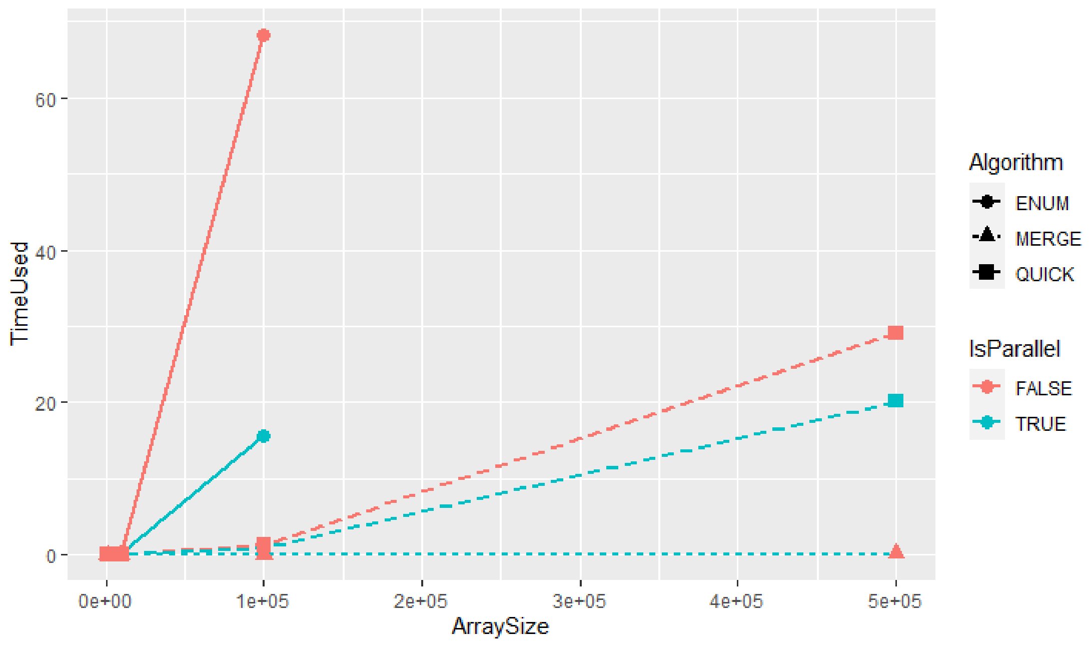
Also, the correctness of each algorithm's parallel solution is tested using a random array with size 10. Serial correctness is tested too, but not screenshotted, because the parallel solution is a more complex solution based on the serial solution. If the parallel one worked correctly, serial should work correctly too.
The original execution screenshots, both the correctness test and speed test are in the attachment file named "ScreenShot.docx"

A result table is generated from the experiment:

Result Table (grouped by array size)

| AS=10^3 | Parallel | Serial |
|---|---|---|
| ENUM | 0.0038 | 0.0053 |
| QUICK | 0.0026 | 0.0003 |
| MERGE | 0.0022 | 0.0002 |
| | | |
| AS=10^4 | Parallel | Serial |
| ENUM | 0.1395 | 0.4749 |
| QUICK | 0.0093 | 0.0125 |
| MERGE | 0.0015 | 0.002 |
| | | |
| AS=10^5 | Parallel | Serial |
| ENUM | 15.6864 | 68.3223 |
| QUICK | 0.8249 | 1.2259 |
| MERGE | 0.0117 | 0.0196 |
| | | |
| AS=5*10^ | Parallel | Serial |
| ENUM | NA | NA |
| QUICK | 20.148 | 29.0929 |
| MERGE | 0.0585 | 0.1007 |

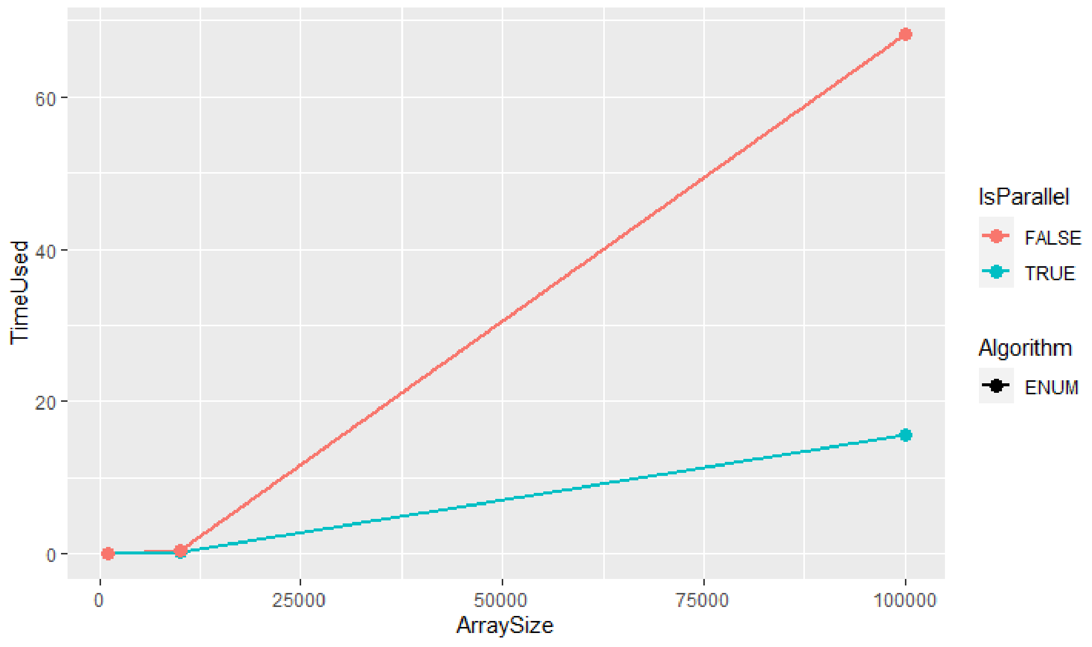
(AS stands for array size, NA means the execution time is unacceptable, it only shows in the enumeration sort when array size is larger than 10^5)
To be more obvious, we can draw an overview graph using RStudio:



From the graph we can obtain that the enumeration sort and quick sort's time consumption grow exponentially while merge sort's time consumption grows linear. The merge sort consumes littlest time and enumeration sort consumes the most time.
This is a simple introduction not considering the parallel or serial programming's influence. We'll investigate how parallel or serial programming influence the sorting speed detailed for each algorithm.

## Enumeration Sort Analysis

For enumeration sort, the time consumption vary by array size and type of solution（serial or parallel）, is shown below



From the graph we can conclude that enumeration sort's time consumption grows

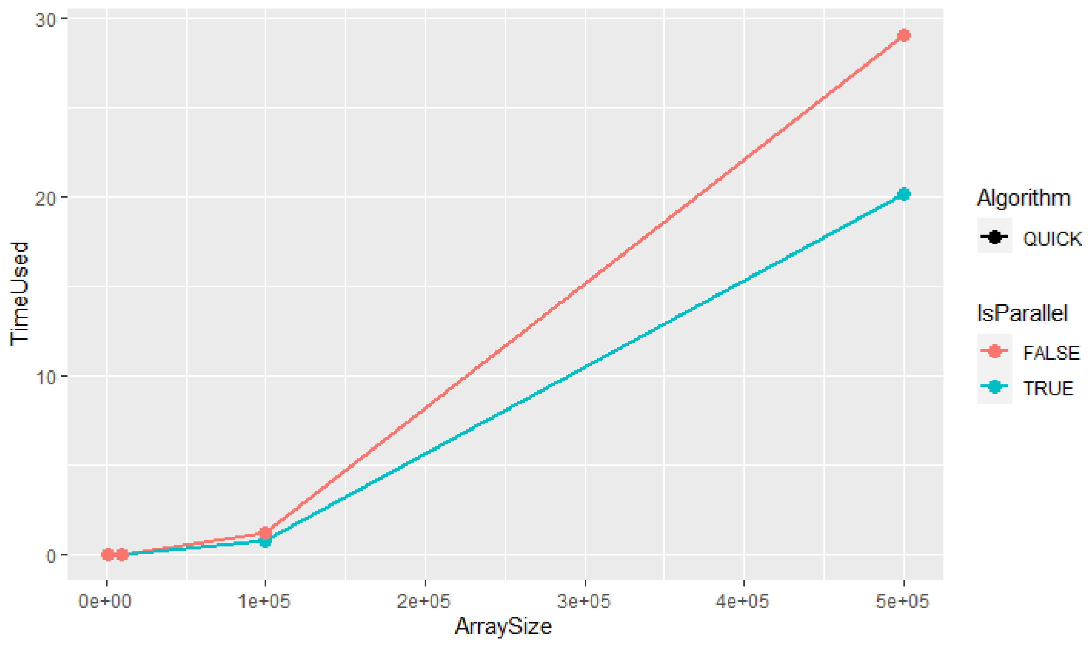
exponentially as the array size grow, despite of what solution is use.
The algorithm contains 3 main parts: 1. Search every item in array and count how many items are smaller than the item. 2. Create a new array then find suitable place in the new array for items using the information obtained before. 3. copying the new array to the original array.
The parallelism can be applied in step 1 and 3, while step 2 is hard to apply parallelism because the loop needs the information from previous round in the same loop (Due to the design of algorithm, it may not be necessary).
The omp parallel function is used to achieve parallelism, because we don't need to carefully decide how loop is divided. Due to the environment, only four processors are available, the omp will divide the for loop to 4 parts. Step 1 and step 3 can reduce 3/4 time it originally costs. Step 2 cannot be speed up, but step 2's time consumption is small and grow slowly too, it could be ignored.
In conclusion, the time consumed in parallel solution should be 1/4 of the serial solution time consumption theoretically. (i.e., the speedup ratio is 4)
In realistic execution, the speed up ratio is calculated below:

| ENUM | Parallel | Serial | SURatio |
|---|---|---|---|
| AS=10^3 | 0.0038 | 0.0053 | 1.3947 |
| AS=10^4 | 0.1395 | 0.4749 | 3.4043 |
| AS=10^5 | 15.6864 | 68.3223 | 4.3555 |

The speedup ratio is smaller until array size gets 10^4, it is because the start for each threads consumes time which is too large to ignore when the total time consumption is slow.
When the array size reaches 10^5, the speedup ratio reaches 4.3555, more the theoretical predict. It should be caused by the little instability generated in the if sentence, every if with a true judgement will cause more steps for the program to do.
Using the same array in two solutions can avoid this problem. But using only one array and renew it every time before using can save more space for the program.

## Quick Sort Analysis

For quicksort, the time consumption vary by array size and type of solution （serial or parallel）,is shown below



In the graph, we can obtain that the time consumption for quick sort also grows exponentially, despite of what solution is use.

We first expected the parallelism applied in quick sort can reduce the sorting time to 1 of n (n the number of processor), but unfortunately, each iteration divides the array into two parts with different length, because sorting stops when both parts are done, the time cost is mainly dependent on the time used to sort larger part of two parts.

For example, if in first iteration, the array is divided to parts that length are 3/4 and 1/4, in serial solution, the program first finish the 3/4 part's sort, then finish the 1/4 part's sort, total time consumption is 1, but in parallel solution, the program start sorting the two parts from the same time, 1/4 part is first sorted but it wait until the 3/4 part is sorted, so the combine time consumption is 3/4, speedup ration should be 1/(3/4), smaller than 2.

In conclude, the speedup ratio for quick sort should be in range 1 to n when using n threads.

In realistic execution, the speedup ratio is calculated below (using 2 threads):

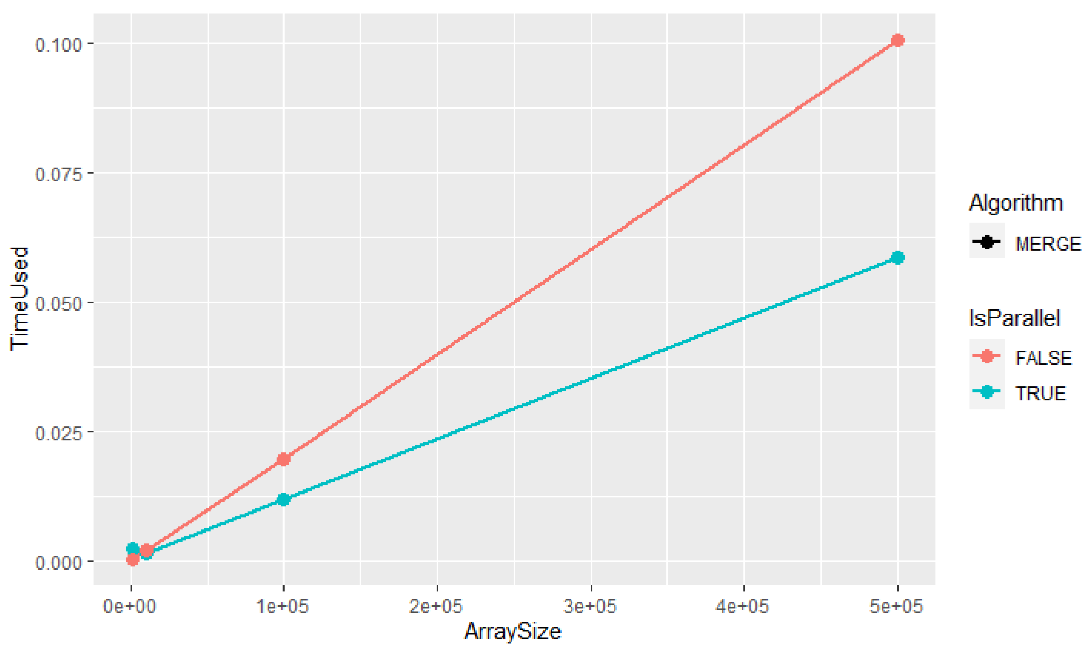
| QUICK | Parallel | Serial | SURatio |
|---|---|---|---|
| AS=10^3 | 0.0026 | 0.0003 | 0.1153 |
| AS=10^4 | 0.0093 | 0.0125 | 1.344 |
| AS=10^5 | 0.8249 | 1.2259 | 1.4861 |
| AS=5*10^ | 20.148 | 29.0929 | 1.4439 |

When array length is 10^3, the speedup ratio is 0.1153 because the start time for threads is significant.

As array size grow, the speedup ratio is close to 1.45 and float above or under it. The cause for this behavior is explained before.

## Merge Sort Analysis

For merge sort, the time consumption vary by array size and type of solution （serial or parallel） , is shown below



In merge sort, because the array is uniformly separated in every round of iterations, the time consumption should be reduced to 1/n (n the number of processor) original time consumption.
That is, the speedup ratio should be 1/2 when using two threads
In realistic execution, the speedup ratio is calculated below (using 2 threads):

| MERGE | Parallel | Serial | SURatio |
|---|---|---|---|
| AS=10^3 | 0.0022 | 0.0002 | 0.0909 |
| AS=10^4 | 0.0015 | 0.002 | 1.3333 |
| AS=10^5 | 0.0117 | 0.0196 | 1.6752 |
| AS=5*10^ | 0.0585 | 0.1007 | 1.7213 |

Like the quick sort, the speedup ratio is low when array size is small, because the start time for threads is significantly large.
The speedup ratio keeps on rising as the array size grows, but only reached 1.7213 when array size is 5*10^5. Is that because the array is not enough so the speedup ratio cannot reach 2?
The answer is yes. Due to the merge sort's time consumption grows linear, the time consumption is still low even the array size is 5*10^5, that means the start time for threads still influences the result significantly. It can be predicted that the speedup ratio will reach 2 if the array size keeps growing

## Part3 Conclusion

The parallel solution can significantly speed up the sorting algorithms.
For enumeration sort, the speedup ratio close to n (n the number of threads).
For quicksort, the speedup ratio changes as the data in array changes, but it should be in range 1 to n.
For merge sort, the speedup ratio can reach n.
Both algorithms' parallel solutions reach the theoretical speedup ratio when the array size is

large enough, or they may be slower than the serial solutions when array size is small.