Assignment 1
(CST435 – Parallel and Cloud Computing)

Program Demo
Github Repo

Jeevittan (146863)
School of Computer Sciences
Universiti Sains Malaysia
11800 USM Penang

1. Algorithm Introduction

The chosen algorithm for this report is Merge sort sorting algorithm. Merge sort algorithm is an algorithm which use the divide and conquer approach in where the algorithm recursively divides the array until it cannot be further divided. This is achieved when the array becomes empty or only have one element in it. Then, the algorithm invokes the merge sort with a bottom to top approach in where 2 set of arrays which was divided previously will be combined once its sorted. Lastly, merge sort have a required auxiliary space of O(n) and time complexity of O(N log(N)) in all cases of worst, average and best.

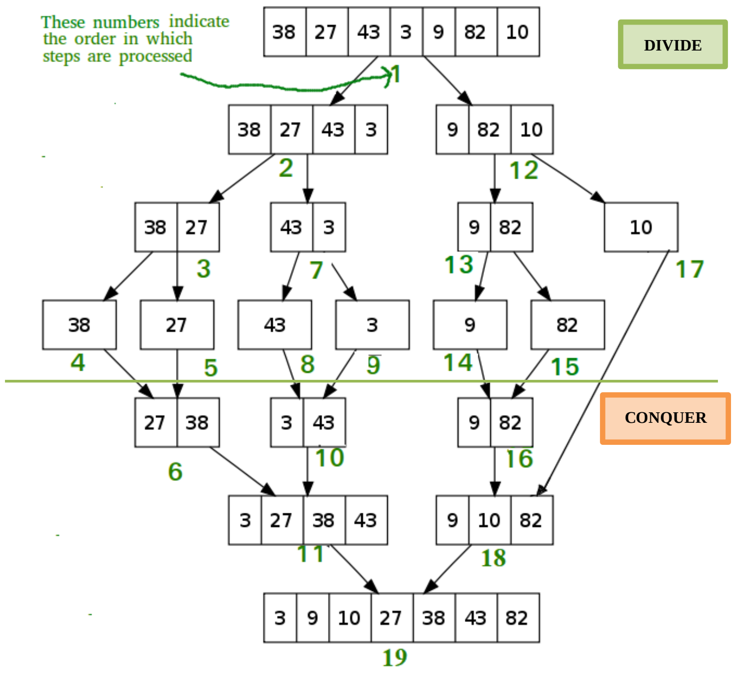
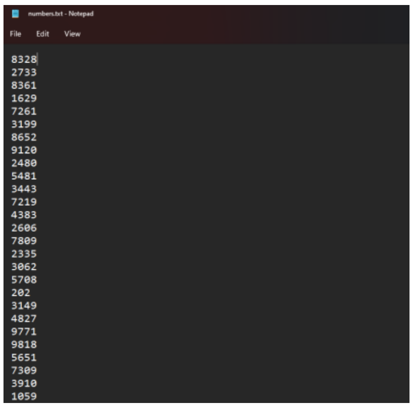

These numbers indicate the order in which steps are processed
DIVIDE
CONQUER
38 27 43 3 9 82 10
3 9 10 27 38 43 82

Figure 1: This is an illustration of how a merge sort works using divide conquer approach

2. Input Format

For this program, there is some requirements for the input file which is converted to an array on numbers and used the input to perform sorting. There is 2 ways an input instance can be created and used:

1. Automatically generate list of numbers in the program by giving the list/array size (N).
2. Use a *name.txt* file which contains a list of numbers. There are some requirements need to be followed for this option:
    a. Only single number per line.
    b. Every value should be separated with a new line (\n).



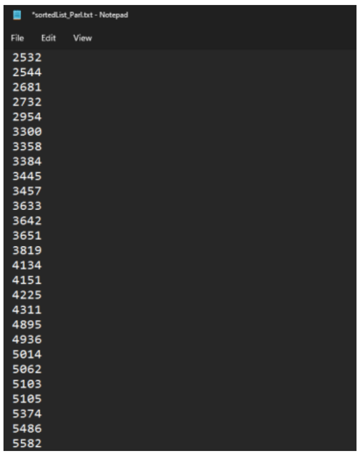
```
8328
2733
8361
1629
7261
3199
8652
9120
2480
5481
3443
7219
4383
2606
7809
2335
3062
5708
202
3149
4827
9771
9818
5651
7309
3910
1059
```

Figure 2: Example of input .txt file format

3. Output Format

Once a sorting algorithm is done executed, the sorted array will be written and be stored as a .txt file in the same directory. The format of the file is very simple which contains 3 content which is:

1. Name of sorting algorithm used
2. Time taken to sort the input array
3. List of sorted numbers (One value per line)



```
2532
2544
2681
2732
2954
3300
3358
3384
3445
3457
3633
3642
3651
3819
4134
4151
4225
4311
4895
4936
5014
5062
5103
5105
5374
5486
5582
```

Figure 3: Example of output .txt file

4. Results

This is the result of Merge Sort algorithm execution using serial and parallel method. The execution is done iteratively by changing the input array size. When the program executes:

1. The array size is entered, array is generated.
2. Serial sorting is done, and sorting time is recorded.
3. Parallel sorting is done, and sorting time is recorded.
4. Step 1-3 is repeated for all other array sizes.
   **Note: Same input array is used for both the series and parallel sorting.*

Size	T_{series}	$T_{parallel}$	Speed-Up	Efficiency	Efficiency (%)	Cost
50	0.001	1.578	0.000634	7.92E-05	0.007921	12.624
100	0.009	1.621	0.005552	0.000694	0.069402	12.968
500	0.004	1.65	0.002424	0.000303	0.030303	13.200
1000	0.006	1.72	0.003488	0.000436	0.043605	13.760
5000	0.023	1.568	0.014668	0.001834	0.183355	12.544
10000	0.047	1.721	0.02731	0.003414	0.341371	13.768
50000	0.25	1.76	0.142045	0.017756	1.775568	14.080
100000	0.569	2.395	0.237578	0.029697	2.969729	19.160
500000	3.26	2.636	1.236722	0.15459	15.45903	21.088
1000000	6.897	3.193	2.160038	0.270005	27.00047	25.544
5000000	40.38	11.498	3.511915	0.438989	43.89894	91.984
10000000	84.983	22.023	3.858829	0.482354	48.23537	176.184

5. Discussion

5.1. Sequential Program Listing

Below are the important code snippets for the execution of serial sorting. There are 2 functions involved in sequential merge sort. Firstly, the *mergeSort(arr)* is the function which accepts an input array and divides the array to left and right subarray until the subarray cannot be further divided. This function's purpose is to recursively divide the input array until the most depth of array is achieved *(array size <=1)*. The given array is dividing to two halves by first identifying the middle element. *(length /2),* then 2 different left and right sub arrays is passed to the *mergeSort* function. Once both the left and right subarray have completely divided until the most depth the array is returned and passed to the *merge(L,R,arr)* function for sorting and conquer stage.

Secondly, the *merge(L,R,arr)* function is used to perform the conquer operation where the sorting is done from the bottom level and then merged the left and right subarrays to form the array of the original size to top root level. Firstly 3 index variables of *i, j, k* is created for tracking the index for 2 subarray and new array during the sorting phase. Next, a recursive sorting until all the array element is accessed in any 2 subarray which is done using while loop. In every iteration of while loop 2 conditions is checked, first is if the current element of left array is smaller than the second array then the smaller element is assigned to the new sorted array. Else it will check the 2nd condition which is to check if the second array element is smaller than the first array element, then the smaller value is assigned to the new sorted array. Once the while loop reaches an end, there will be 2 more while loops to check if there's any missed-out elements from any of the 2 sub arrays. If there is then it will be assigned to the sorted array. The maximum possible element to be missed is only 1, therefore it is not required to check if the element is smaller than any other elements. Once all elements is accessed and sorted the function returns an array.

```
def merge(L,R,arr):
    i = j = k = 0
    # copy data into temporary array named L and R
    while i < len(L) and j < len(R):
        if L[i] < R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
    # check if there is any element left
    while i < len(L):
        arr[k] = L[i]
        i += 1
        k += 1
    while j < len(R):
        arr[k] = R[j]
        j += 1
        k += 1
    return arr
```

Snippet 4.1.1: Merge function which is used to combine the divided arrays.

```
def mergeSort(arr):
    if len(arr) > 1:
        # finds the middle element of the array
        mid = len(arr) // 2
        # splits array elements into 2 parts (Left & Right)
        L = arr[:mid]
        #print(L)
        R = arr[mid:]
        #print(R)
        # Sort first half
        mergeSort(L)
        # Sort second half
        mergeSort(R)
        return merge(L,R,arr)

    else:
        return arr
```

Snippet 4.1.2: Merge Sort function which is used to divide and sort the array into subarrays.

5.2.Parallel Program Listing

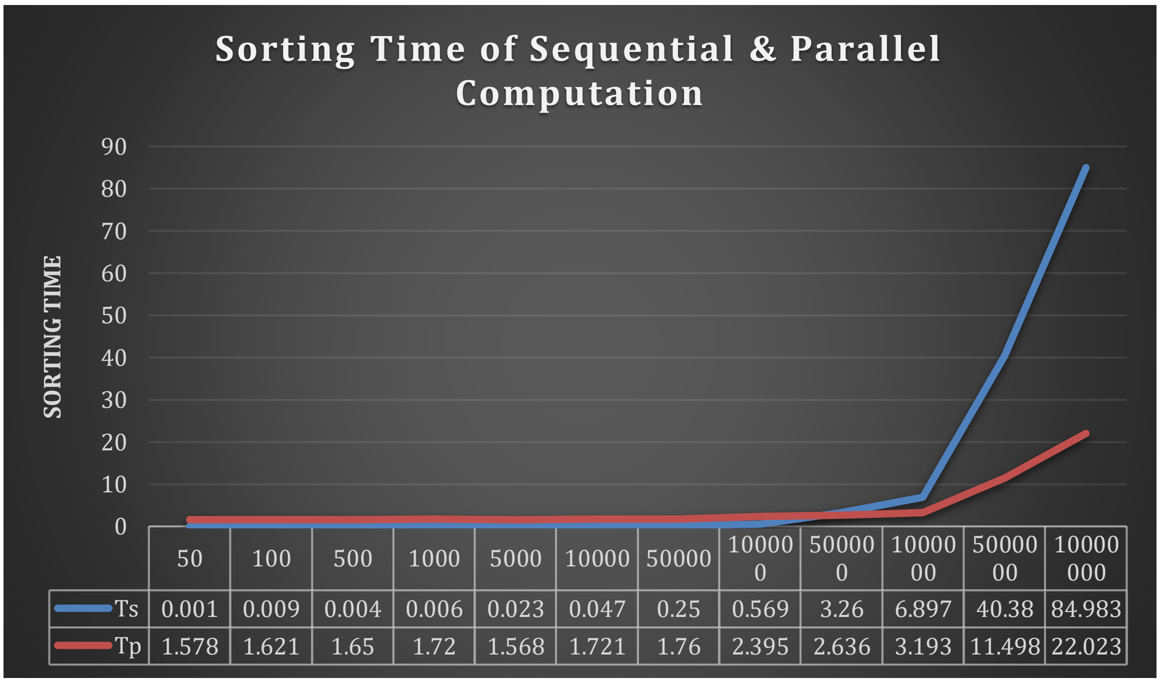
As for the parallel method of merge sort, the additional function *parallelMergeSort* is used besides the *mergeSort* and *merge* function. The *parallelMergeSort* function is used to allocate each processor of CPU to executes a defined set of tasks which is the sorting. Firstly, the processor count is checked where if it is 1 then a serial merge sort is done or else parallel merge sort is done recursively. Two package is used to implement parallel computation which is *multiprocessing* and *concurrent* where the concurrent is mainly used for the parallel computation. The *futures.ProcessPollExecutor* is a programming pattern which automatically manages a pool of worker processes such as controlling the creation of the pool when it is needed and dynamic load balancing when the worker processor is idle. This is done by allocating half of existing pool processor for each division (left & right). Hence several computations occur concurrently. For example when there is 8 available cores the *p.map()* function will iterate until there is 1 or less pool of processors is left. Hence at the end of the map operation each core will have a subarray to be sorted using the *merge* function. The object returned from the parallel processing is stored in the future variable as an array of subarrays and hence it is passed to the merge function for the sorting and conquer phase.

```
def parallelMergeSort(arr,cpu_count=multiprocessing.cpu_count()):
# By default, unless cpu count is entered specifically, it takes the number of processors
from the system.
# run until the number of processors is reduced to 1
# halve the number of processors. (2 processors go to each pool) Determines how many more
times it can enter
# For example, if the number of processors is 16, it can enter 2 to 4 times, while reaching
a depth of 4 units, it runs 16 processes.
# Since pool expects an iterable value, we add a separate list to the left and right lists.
        mid = len(arr) // 2
        left = arr[:mid]
        right = arr[mid:]
        full = [left, right]
        if cpu_count == 1:
            l = mergeSort(left)
            r = mergeSort(right)
            return merge(l,r,arr)

        else:
            result = []
            with futures.ProcessPoolExecutor(cpu_count) as p:
                cpu_count = cpu_count // 2
                if cpu_count > 0:

                    cpu_countlist = [cpu_count,cpu_count]
                    future = p.map(parallelMergeSort,full,cpu_countlist)
                    for value in future:
                        result.append(value)

            return merge(result[0],result[1],arr)
```

Snippet 5.1.1:Parallel Merge Sort function for parallel execution of sorting.

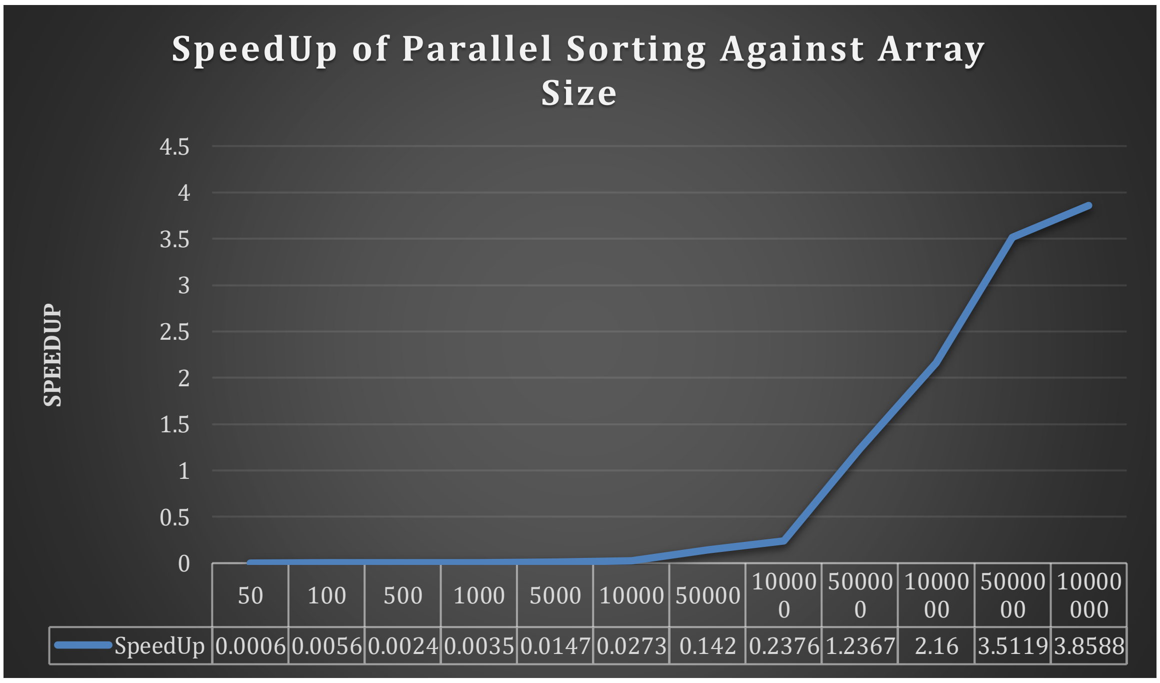
5.3.Scalability Analysis

1. The first analysis done is the analysis of sorting time for both sequential and parallel computation using 12 different array sizes. It is observed that the parallel processing took comparatively shorter time to do the sorting for array size range of 500000 – 10000000.



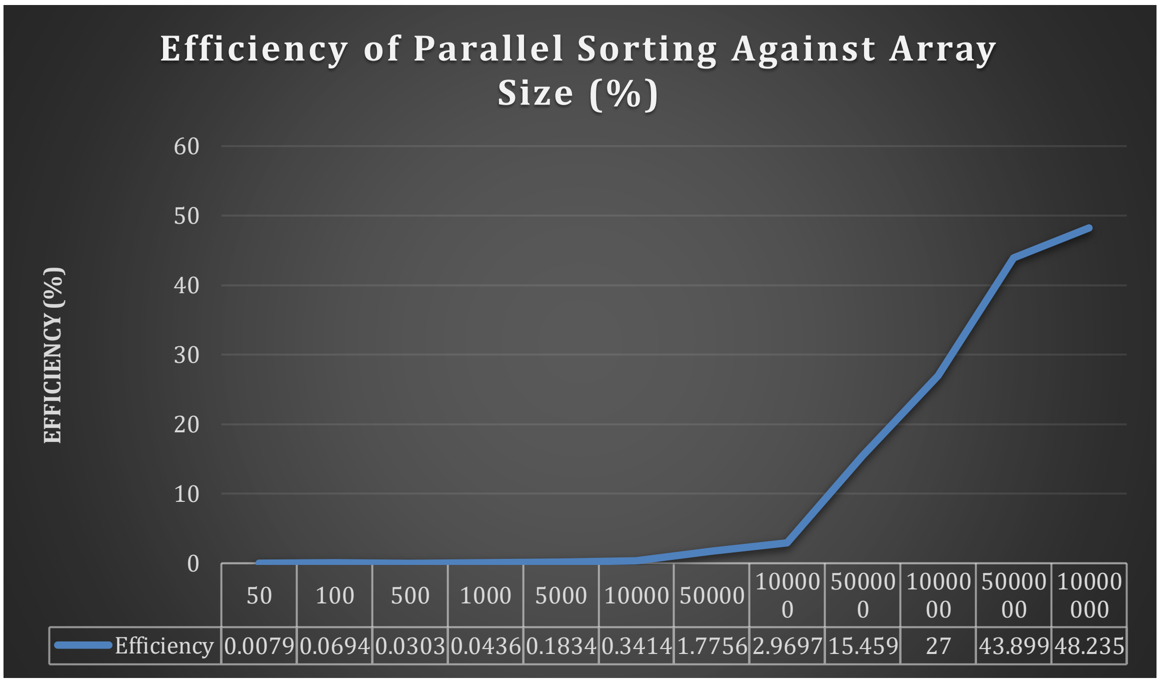
	50	100	500	1000	5000	10000	50000	100000	500000	1000000	5000000	10000000
Ts	0.001	0.009	0.004	0.006	0.023	0.047	0.25	0.569	3.26	6.897	40.38	84.983
Tp	1.578	1.621	1.65	1.72	1.568	1.721	1.76	2.395	2.636	3.193	11.498	22.023

2. The second analysis is the speedup analysis of parallel sorting against array size. The formula for Speed-up is [(Ts)/(Tp)]. From the analysis an increase in speedup is observed starting from input size above 10000. An non-linear rate of speedup can be seen at input size of >500000.



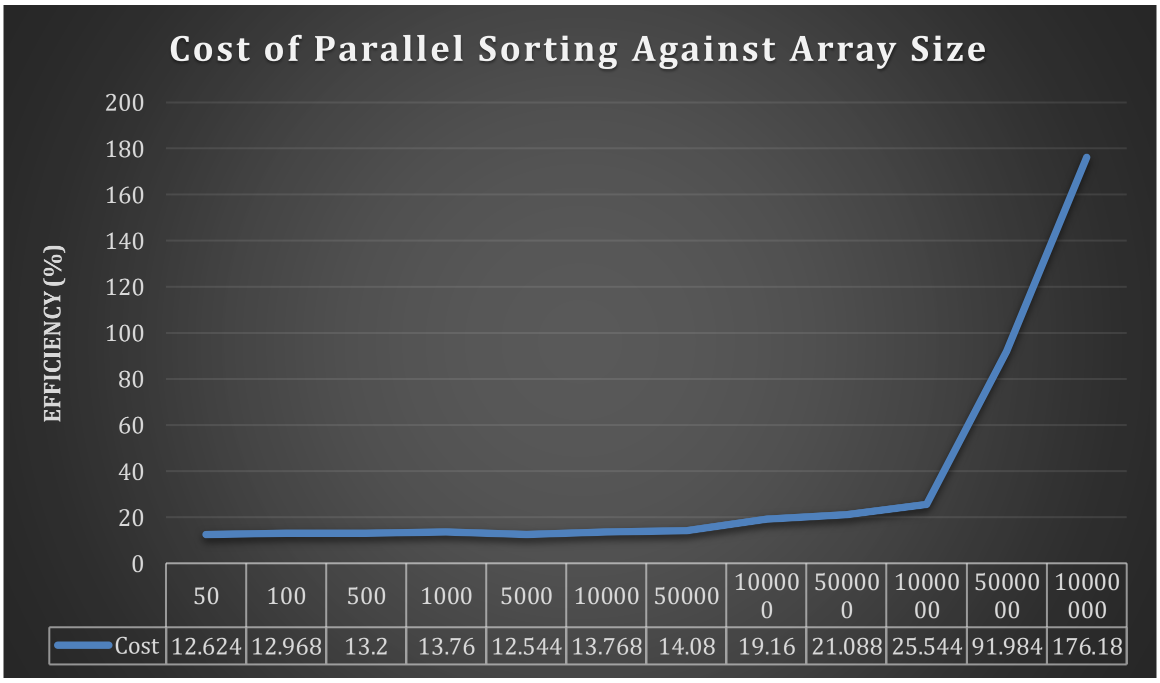
	50	100	500	1000	5000	10000	50000	100000	500000	1000000	5000000	10000000
SpeedUp	0.0006	0.0056	0.0024	0.0035	0.0147	0.0273	0.142	0.2376	1.2367	2.16	3.5119	3.8588

3. The third analysis is the efficiency of parallel sorting (%) against array size. The formula for efficiency is *[(Speedup / P)*100] %*. A similar pattern of Speed-up graph can be observed here too where the parallel computation has a linear increase from input size 100000 to 10000000.



	50	100	500	1000	5000	10000	50000	100000	500000	1000000	5000000	10000000
Efficiency	0.0079	0.0694	0.0303	0.0436	0.1834	0.3414	1.7756	2.9697	15.459	27	43.899	48.235

4. The fourth analysis is the cost against array size analysis, where the formula used is *[T(P) * P]*. From the analysis it is observed a quick rise in sorting cost from array size range of 100000 – 10000000.



	50	100	500	1000	5000	10000	50000	100000	500000	1000000	5000000	10000000
Cost	12.624	12.968	13.2	13.76	12.544	13.768	14.08	19.16	21.088	25.544	91.984	176.18

6. Conclusion & Lesson Learned

As a conclusion, both sequential and parallel approach have its own pros and cons considering a few factors that might affect the sorting time and cost of the algorithm execution such as load balancing, communication, and synchronization. For an example, a parallel approach only outperforms the sequential when the given input is around > 500000. Hence, parallel method is not always fast as what Amdahl's law have stated in every parallel execution certain portion will be bounded by serial execution. Therefore, following the Gustafson's law can make sure an efficient execution by using provided resources efficiently. This can be done by manipulating the input size to make a coarse grain size.

7. References

[1] GeekforGeeks. (23 Sep, 2022). Merge Sort Algorithm. Retrieved from https://www.geeksforgeeks.org/merge-sort

[2] mertcenkk. (14 May, 2021). Merge Sort with Parallel Programming. Retrieved from mertcenkk/ParallelMergeSort: Merge sort with parallel programming (github.com)

[3] Jason Brownlee. (26 Jan 2022). ProcessPoolExecutor in Python: The Complete Guide. Retrieved from ProcessPoolExecutor in Python: The Complete Guide (superfastpython.com)