PROJECT 1 REPORT

Part A: Pseudo-codes:

Quick Sort - Serial solution:

```
algorithm quickSort(array, low, high):
    // Low and high are inclusive: meaning at start low would be 0 and high would be
    // array length - 1
    // Check if indexes are natural and in correct order
    if low >= 0 && high >= 0 && low < high:
        // Choose pivot, lowest value
        pivot = low
        // Start sorting within specify section
        for i = low+1 to high:
            if array[i] <= array[pivot]:
                move array[i] to before array[pivot]
                pivot++

        // Recursion call on the left side and the right side of the array
        if pivot > low:
            quickSorting(array, low, pivot – 1)
        if pivot < high:
            quickSorting(array, pivot + 1, high)
```

Quick Sort – Parallel solution:

```
algorithm paraQuickSort(array, low, high):
    // Low and high are inclusive: meaning at start low would be 0 and high would be
    // array length - 1
    // Check if indexes are natural and in correct order
    if low >= 0 && high >= 0 && low < high:
        // Choose pivot, lowest value
        pivot = low
        // Start sorting within specify section
        for i = low+1 to high:
            if array[i] <= array[pivot]:
                move array[i] to before array[pivot]
                pivot++

        // Parallel call on the left side and the right side of the array
        if pivot > low:
            fork:
                quickSorting(array, low, pivot – 1) )

        if pivot < high:
            quickSorting(array, pivot + 1, high)

        join
```

Merge Sort – Serial solution:

```
// Implement bottom-up merge sort
algorithm buMergeSort(array):
        for width = 1; width < array.length; width = 2 * width:
                for index = 0; index < array.length; index = index + (2 * width):
                        indexA = index
                        indexB = index + width

                        if indexB < array.length:
                                endIndex = min(index + (2 * width) , array.length)

                        // Pass through to merging array
                        mergeArray(array, indexA, indexB, endIndex)

algorithm mergeArray(array, indexA, indexB, endIndex):
        // Create 2 sub array A and B from array
        lengthA = indexB – indexA
        lengthB = endIndex – indexB
        arrayA = array[indexA : indexB]
        arrayB = array[indexB : endIndex]

        // Indexes to iterate through arrayA and arrayB
        iA = 0
        iB = 0
        for i = indexA to i < endIndex:
                // Check for out of bound
                if iA >= lengthA:
                        array[i] = arrayB[iB]
                        iB++
                else if indexB >= lengthB:
                        array[i] = arrayA[iA]
                        iA++
                else:
                        // General cases
                        if arrayA[iA] <= arrayB[iB]:
                                array[i] = arrayA[iA]
                                iA++
                        else:
                                array[i] = arrayB[iB]
                                iB++
```

Merge Sort – Parallel solution:

```
// Implement bottom-up merge sort
algorithm paraBUMergeSort(array):
        for width = 1; width < array.length; width = 2 * width:
                for index = 0; index < array.length; index = index + (2 * width):
                        indexA = index
                        indexB = index + width

                        if indexB < array.length:
                                endIndex = min(index + (2 * width) , array.length)

                        // Fork a parallel and pass through to merging array
                        fork:
                                 mergeArray(array, indexA, indexB, endIndex)
                join

algorithm mergeArray(array, indexA, indexB, endIndex):
        // Create 2 sub array A and B from array
        lengthA = indexB – indexA
        lengthB = endIndex – indexB
        arrayA = array[indexA : indexB]
        arrayB = array[indexB : endIndex]

        // Indexes to iterate through arrayA and arrayB
        iA = 0
        iB = 0
        for i = indexA to i < endIndex:
                // Check for out of bound
                if iA >= lengthA:
                        array[i] = arrayB[iB]
                        iB++
                else if indexB >= lengthB:
                        array[i] = arrayA[iA]
                        iA++
                else:
                        // General cases
                        if arrayA[iA] <= arrayB[iB]:
                                array[i] = arrayA[iA]
                                iA++
                        else:
                                array[i] = arrayB[iB]
                                iB++
```

Enumeration Sort – Serial solution:

```
algorithm enumSort(array):
    returnAr[array.length]

    for i = 0 to i < array.length:
        rank = 0
        dupe = 0
        for j = 0 to j < array.length:
            if i != j:
                if array[i] > array[j]:
                    rank++
                else if array[i] == array[j]:
                    dupe++

        // Put values to its corresponding rank
        // Check if the value been put before
        if returnAr[rank] != array[i]:
            for k = 0 to dupe:
                returnAr[rank + k] = array[i]
```

Enumeration Sort – Parallel solution:

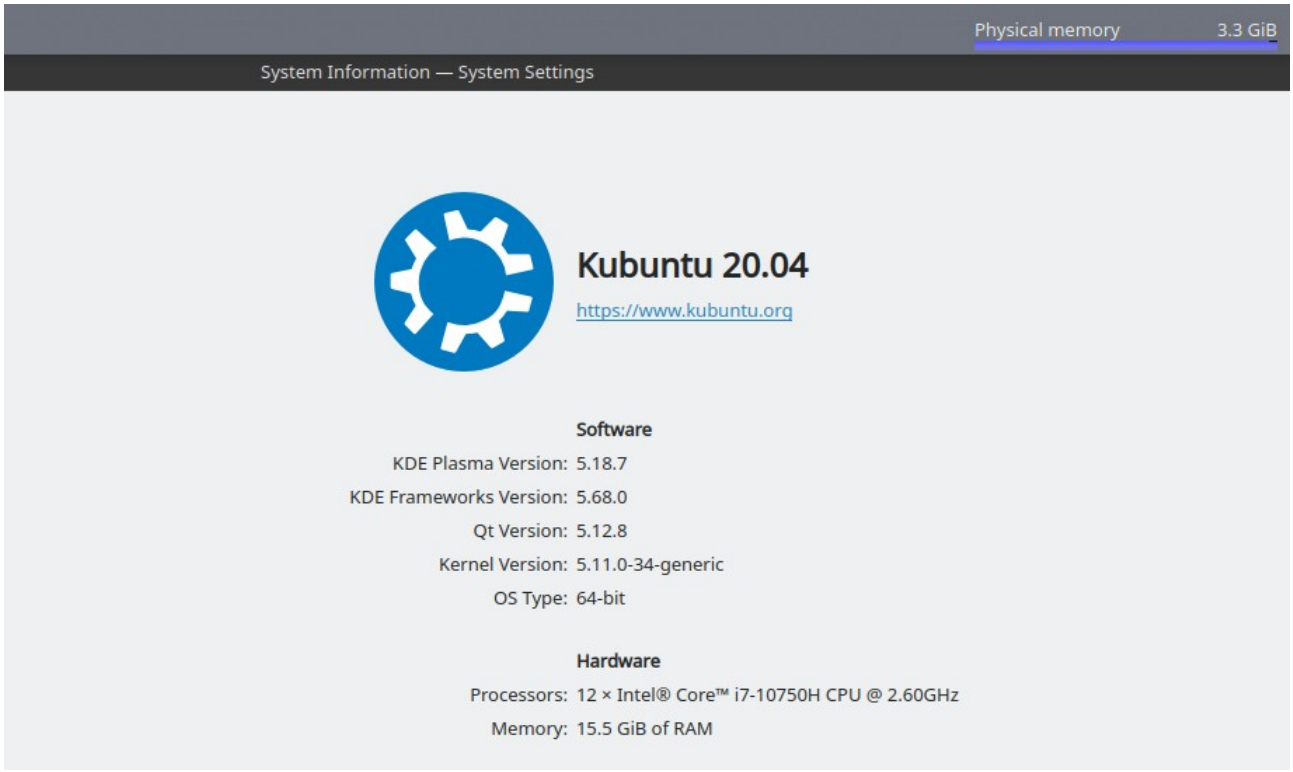
```
algorithm paraEnumSort(array):
    returnAr[array.length]

    for i = 0 to i < array.length:
        fork:
            rank = 0
            dupe = 0
            for j = 0 to j < array.length:
                if i != j:
                    if array[i] > array[j]:
                        rank++
                    else if array[i] == array[j]:
                        dupe++

            // Put values to its corresponding rank
            // Check if the value been put before
            if returnAr[rank] != array[i]:
                for k = 0 to dupe:
                    returnAr[rank + k] = array[i]
    join
```

Part B: Description of Experiment Environment:

- CPU: 12 cores Intel i7
- RAM: 15.5 gigabyte
- Operating System: Kubuntu 20.04 (64 bit)



Part C: How to compile source code:

*** Notice:**

- My compilation and usage for project 1 are made with my understanding that there will be 3 users inputs, these include: number of threads, array size/length, and specific algorithm to run.
- The code **will** print out the run time and check the output of every sorting algorithm returns against the output of library sort function "qsort"

To compile:

+ Within folder "project1code" there is a "makefile". It contains command to compile the main "project1.c" file and every other satellite files for the entire project
+ Open terminal within "project1code" and write "make", the compiled out file is set to be "sort"

To run project 1:

+ After called "make" the executable file is "sort"
+ There are 2 options to run "sort":
  - Option 1: write in terminal "./sort" : This will run the default settings with: 8 threads, 10 000 000 array length, and action "d" means run serial and parallel sort algorithm of Merge Sort and Quick Sort.
  - Option 2: Specify inputs, **must** full-fill all 3 inputs:

  ./sort {Number of threads} {Array length} {Action}

  Number of threads: Specify the number of threads for parallel runs

  Array length: Specify the array size to be randomly generated

  Action: One of 4 options, all will run serial and parallel solutions for each sort:
    "default" for merge sort and quick sort run
    "merge" for merge sort only run
    "quick" for quick sort only run
    "enum" for enumeration sort only run
    "all" for all sort algorithms run

Part D: Experimental results:

*** Notice:**

- The results in tables below are the calculated average of 20 looping runs(Arrays values are randomised, not the length), unless specify otherwise.
- Times are in seconds.
- Percentage reduction (%): Calculated by ((SerialTime – ParallelTime) / SerialTime) * 100.

Table of running time for settings 1: 8 threads, 1 000 array size:

	Merge Sort	Quick Sort	Enumeration Sort
Serial Solution	0.0001485500	0.0001168500	0.0054243500
Parallel Solution	0.0000924000	0.0010199500	0.0010039500
Percentage Reduction	37.7987209694	-772.8712023962	81.4917916432

Table of running time for settings 2: 8 threads, 10 000 array size:

	Merge Sort	Quick Sort	Enumeration Sort
Serial Solution	0.0013982500	0.0011946500	0.4469627000
Parallel Solution	0.0003843000	0.0039715500	0.0881076500
Percentage Reduction	72.5156445557	-232.4446490604	80.2874714154

Table of running time for setting 3: 8 threads, 25 000 array size:

	Merge Sort	Quick Sort	Enumeration Sort
Serial Solution	0.0033723000	0.0028836000	2.5617805500
Parallel Solution	0.0008819000	0.0038358000	0.5192160500
Percentage Reduction	73.8487085965	-33.0212234707	79.7322198422

Table of running time for settings 4: 8 threads, 50 000 array size

	Merge Sort	Quick Sort	Enumeration Sort
Serial Solution	0.0070512500	0.0063924000	10.7647558000
Parallel Solution	0.0019102000	0.0052267000	1.9328049500
Percentage Reduction	72.9097677717	18.2357174144	82.0450645987

Table of running time for settings 5: 8 threads, 100 000 array size:

	Merge Sort	Quick Sort	Enumeration Sort
Serial Solution	0.0151284500	0.0139563000	44.4280114000
Parallel Solution	0.0040317500	0.0083246000	7.6520645000
Percentage Reduction	73.3498805231	40.3523856610	82.7764865929

Table of running time for settings 6: 8 threads, 1 000 000 array size, no enumeration sort:

	Merge Sort	Quick Sort
Serial Solution	0.1680038500	0.1460736000
Parallel Solution	0.0422331000	0.0501359000
Percentage Reduction	74.8618260832	65.6776446942

Table of running time for settings 7: 8 threads, 10 000 000 array size, no enumeration sort:

	Merge Sort	Quick Sort
Serial Solution	2.3431978500	1.9429952500
Parallel Solution	0.6351149500	0.5190189500
Percentage Reduction	72.8953767178	73.2876881711

Table of running time for settings 8: 8 threads, 100 000 000 array size, no enumeration sort:

	Merge Sort	Quick Sort
Serial Solution	26.6471898500	21.9411191000
Parallel Solution	6.7622267500	5.2042670000
Percentage Reduction	74.6231149023	76.2807586237

Part D: Speed and Time Analysis:

- At 8 threads settings, there is a significant reduction in time from Serial Solution to Parallel solution:
    + Merge sort: Average reduction range between 72 – 75% with sufficient array length
    + Quick sort: Average reduction range more wildly between 65 – 76% with sufficient array length
- One thing of noted: When the array length are smaller, then time differences start to reduce. With Quick Sort Parallel Solution even slower than its serial solution, this can be understand as the small array length allows the Serial Solution run time not significant enough to justify the time Thread Pool task distribution, making the Quick Sort Parallel Solution slower than Quick Sort Serial Solution.
- The run speed for Parallel Solution Sorting algorithm significantly faster with the increase in array length