PROJECT 1 REPORT

Part A: Pseudo-codes:

SORTING ALGORITHMS 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)
```

- Explanation:

+ Choose the first element of the array as the pivot and perform partition upon the array with smaller value than pivot on left side and greater value on right side

+ Decide 2 sub section divided by the pivot with left section going from the pivot to the start and the right section going from the pivot to the end (pivot **exclusive**)

+ Pass those 2 section into recursive loop until the small sub section of only one value available, then return

+ The algorithm will serially run the left side until completion before it attempt to run right side of any sub section

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
```

- Explanation:
    + Choose the first element of the array as the pivot and perform partition upon the array with smaller value than pivot on left side and greater value on right side
    + Decide 2 sub section divided by the pivot with left section going from the pivot to the start and the right section going from the pivot to the end (pivot **exclusive**)
    + Fork one section recursive call into another thread, run one section recursive loop in the current thread until the small sub section of only one value available, then return
    + The algorithm will run parallel the left side and right side of any sub section if there are enough threads in the thread pool

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++
```

- Explanation:

+ Loop through sub array list width: Starting with width = 1 and increasing it to width = 2 * width after each loop

+ Create 2 sub array with the decided width above by looping through the starting index of the total array, make sure no sub array would references values outside of the total array bound

+ Perform merging array of the 2 sub array created above

+ The algorithm performs merging array serially until the end of the total array then the sub array width loop increases

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++
```

- Explanation:
    + Loop through sub array list width: Starting with width = 1 and increasing it to width = 2 * width after each loop
    + Create 2 sub array with the decided width above by looping through the starting index of the total array, make sure no sub array would references values outside of the total array bound
    + Perform merging array of the 2 sub array created above
    + The algorithm performs merging array in parallel for each pair of sub array then the sub array width loop increases

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]
```

- Explanation:
    + The algorithm choose a value from the total array
    + Then it compares and track how many other values that are smaller and how many are equal
    + Then it place that chosen value within the sorted array and place any number of duplicates after the chosen value within the sorted array.
    + The function serially repeatedly chooses the next value from the array until finished

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
```

- Explanation:
    + The algorithm choose a value from the total array
    + Then it compares and track how many other values that are smaller and how many are equal
    + Then it place that chosen value within the sorted array and place any number of duplicates after the chosen value within the sorted array.
    + The function repeatedly chooses the next value from the array until finished in parallel, meaning it does the comparison of each chosen value simultaneously in different threads.

MPI parallel sorting algorithm:

*** Note:** The MPI sorting algorithm are similar in design as it simply divide the original array into mostly equal size sub array and sort it within each processes. The core working (sorting work within each processes) are identical to OMP parallel design above. After every process finished sorting, perform the merging of every sub array with the total sorted array exist only in root process.

```
algorithm mpiSort(array[], totalSize, eachSubArraySize[], sortAlgoToRun):
        rank = getMPIRank()
        mpiSize = getMPISize()

        subArraySize = eachSubArraySize[rank]
        startAt = rank * eachSubArraySize[0]

        subArray = array[startAt : (startAt + subArraySize)]

        if ( sortAlgoToRun == “quick”):
                subArray = paraQuickSort(subAr, startAt, startAt + subArraySize)
        else if ( sortAlgoToRun == 'merge'):
                subArray = paraBUMergeSort(subAr)
        else if ( sortAlgoToRun == 'enumeration'):
                subArray = paraEnumSort(subAr)
        else:
                exit(EXIT_FAILURE)

        currentLen = sortLen
        for(int step = 1; step < parallelNum; step = 2 * step):
                if (rank % (2 * step) != 0):
                        sendTo = rank – step
                        MPI_Send(&currentLen, 1, MPI_INT, sendTo, 0, MPI_COMM_WORLD);

                        MPI_Send(subAr, currentLen, MPI_DOUBLE, sendTo, 0,
MPI_COMM_WORLD)
                        break
                if (rank + step < parallelNum):
                        recvFrom = rank + step

                        recvSize;
                        MPI_Recv(&recvSize, 1, MPI_INT, recvFrom, 0, MPI_COMM_WORLD,
&status);
                        recvAr[];
                        MPI_Recv(recvAr, recvSize, MPI_DOUBLE, recvFrom, 0,
MPI_COMM_WORLD, &status);

                        subArray = mergeArray(subAr, recvAr, currentLen, recvSize);
                        currentLen += recvSize
```

```
    // Finished merging every array
    // Only root process has the full sorted array
    MPI_Barrier(MPI_COMM_WORLD);

    return subArray
```

- Explanation:
    + Every processes holds the total array, then each calculate the sub array that it is responsible for sorting
    + Then, each processes call the sorting algorithm decided by the sortAlgoToRun var and run the according sorting algorithm, I chose to run OMP parallel version of each sorting algorithms
    + After every processes have finished sorting, start merging every sub arrays with the end goal of having the fully sorted array in root processes.

IO OPERATIONS PSEUDO CODES:

```
algorithm serialWrite(array, arraySize, fileName):
    file = fopen(fileName, “write” “binary” mode)
    fwrite(array, array value type, arraySize, file)

    fclose(file)
```

- Explanation:
    + Write to binary file the array with specified array length and each array value type into the specified file name

```
algorithm serialRead(array, arraySize, fileName):
    // Check if file exist
    if access(fileName, F_OK) == 0:
        file = fopen(fileName, “read” “binary” mode)
        fread(array, array value type, arraySize, file)

        fclose(fp)
```

- Explanation:
    + Read the content from the binary file into the specified array with specified array length and each array value type in the specified file name

```
algorithm mpiWrite(array, arraySize, fileName):
    file = open file in MPI write mode(fileName)
    averageLen = arraySize / mpi world size
    startAt = current process rank * averageLen

    if(current process rank is last in the mpi world):
        averageLen = arraySize – (startAt * averageLen)
```

“MPI File write at” function(file, startAt * sizeof(array value type), array + startAt, averageLen, MPI type for “array value type”);
close MPI file()

- Explanation:
+ With the write operation in MPI, each processes are assigned a mostly equal size portion of the total array to write into the specified file name. This writing action is done in parallel.

algorithm mpiRead(array, arraySize, fileName):
// Check if file exists
if access(fileName, F_OK) == 0:
file = open file in MPI write mode(fileName)

averageLen = arraySize / mpi world size
for i = 0 to < mpi world size
subArraySize[i] = averageLen
displacements[i] = startAt * averageLen
subArraySize[mpi world size – 1] = arraySize – displacements[mpi world size - 1]

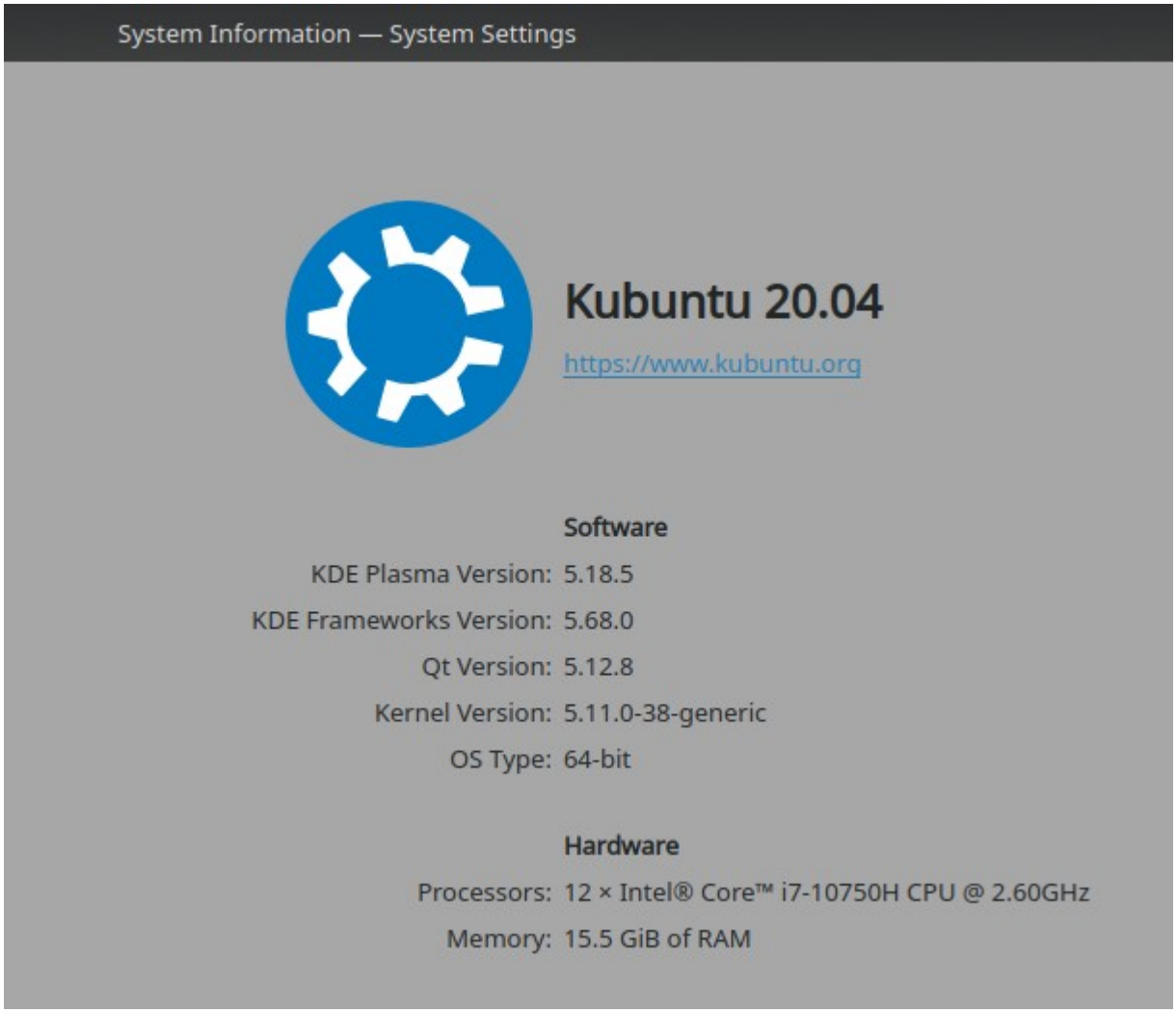
“MPI File read at” function(file, displacements[rank] * sizeof(array value type), array + displacements[rank], subArraySize[rank], MPI type for “array value type”)

// Perform MPI_Gather to get the full array in the root process
if current process is root:
“MPI Gatherv” function(reArr + displacements[rank], subArraySize[rank], MPI type for “array value type”, reArr, subArraySize, displacements, MPI type for “array value type”, 0, MPI_COMM_WORLD)
else:
“MPI Gatherv” function(reArr + displacements[rank], subArraySize[rank], MPI type for “array value type”, NULL, NULL, NULL, MPI type for “array value type”, 0, MPI_COMM_WORLD)
close MPI file()

- Explanation:
+ With read operation in MPI, each processes are assigned a mostly equal size portion of the file to read into each sub array buffer. Afterwards, we use MPI_Gatherv to gather all sub array into a full array store in the root process.

Part B: Description of Experiment Environment:

- CPU: 6 cores 12 cpu Intel i7
- RAM: 15.5 gigabyte
- Operating System: Kubuntu 20.04 (64 bit)



Part C: How to compile source code:

*** Notice:**

- 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 “project2code” there is a “makefile”. It contains command to compile the main “mpiMain.c” file and every other satellite files for the entire project
+ Open terminal within “project2code” and type “make”, the compiled out file is set to be “main”

To run project 2, type:

make {run option} process={int value here} thread={int value here} size={int value here} action={string value here}

NOTE: You must type make and {run option}. The others value are optional and if empty will default to default values

+ run option: determine how the executable program will run
    * One of 4 options:
        run: to run the general program of IO operations and sorting algo with the specify parameters below
        reportIO: to run only IO operations in loop of 10 times and display the average time each and every operations
        report: to run only the Sorting algos in loop of 10 times with the specify parameters below
        runsmall: to run the general program like "run" but with size=10001 and action=all
+ process: determine the number of process for mpiexec run (default : 4)
+ thread: determine max number of omp thread each process will run (default : 2)
+ size: determine the array size to be generated and sorted (default : 10000001)
+ action: determine which sorting algorithm will run: (default : default)
    * One of 4 options, all will run serial and mpi 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 10 looping runs and 2 omp threads(Arrays values are randomised, not the length), unless specify otherwise.
- Times are in seconds.
- Percentage reduction (%): Calculated by ((SerialTime – ParallelTime) / SerialTime) * 100.

SORTING ALGORITHMS EXPERIMENTAL RESULTS:

Table of running time for settings 1: 4 processes, 1 000 array size:

	Merge Sort	Quick Sort	Enumeration Sort
Serial Solution	0.0001256000	0.0000906000	0.0048109000
MPI Solution	0.0001397000	0.0001679000	0.0002532000
Percentage Reduction	-11.2261146497	-85.3200883002	94.7369515060

Table of running time for settings 2: 4 processes, 10 000 array size:

	Merge Sort	Quick Sort	Enumeration Sort
Serial Solution	0.0014280000	0.0012004000	0.5162433000
MPI Solution	0.0010585000	0.0008030000	0.0241057000
Percentage Reduction	25.8753501401	33.1056314562	95.3305544111

Table of running time for setting 3: 4 processes, 25 000 array size:

	Merge Sort	Quick Sort	Enumeration Sort
Serial Solution	0.0034042000	0.0029781000	2.8181579000
MPI Solution	0.0010676000	0.0014315000	0.1370312000
Percentage Reduction	68.6387403795	51.9324401464	95.1375613127

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

	Merge Sort	Quick Sort	Enumeration Sort
Serial Solution	0.0074000000	0.0064717000	11.7307933000
MPI Solution	0.0020273000	0.0024765000	0.5203634000
Percentage Reduction	72.6040540541	61.7333930806	95.5641243802

Table of running time for settings 5: 4 processes, 100 000 array size:

	Merge Sort	Quick Sort	Enumeration Sort
Serial Solution	0.0155425000	0.0137547000	48.7166845000
MPI Solution	0.0041272000	0.0045250000	2.0149106000
Percentage Reduction	73.4457133666	67.1021541728	95.8640235462

Table of running time for settings 6: 4 processes, 1 000 000 array size, no enumeration sort:

	Merge Sort	Quick Sort
Serial Solution	0.1954371000	0.1723442000
MPI Solution	0.0483033000	0.0522628000
Percentage Reduction	75.2844777169	69.6753357525

Table of running time for settings 7: 4 processes, 10 000 000 array size, no enumeration sort:

	Merge Sort	Quick Sort
Serial Solution	2.3631138000	2.0500279000
MPI Solution	0.5633098000	0.5418920000
Percentage Reduction	76.1623921793	73.5666036545

Table of running time for settings 8: 4 processes, 100 000 000 array size, no enumeration sort:

	Merge Sort	Quick Sort
Serial Solution	28.1227420000	21.4965390000
MPI Solution	6.1827500000	5.8634670000
Percentage Reduction	78.015124	72.723670

IO OPERATIONS EXPERIMENTAL RESULTS:

Table of running time for settings 1: 4 processes, 1 000 array size

	Write operation	Read operation
Serial Solution	0.0000618000	0.0000077000
MPI Solution	0.0003752000	0.0002352000
Percentage Reduction	-507.1197411003	-2954.5454545455

Table of running time for settings 2: 4 processes, 10 000 array size

	Write operation	Read operation
Serial Solution	0.0000932000	0.0000130000
MPI Solution	0.0003475000	0.0002196000
Percentage Reduction	-272.8540772532	-1589.2307692308

Table of running time for settings 3: 4 processes, 100 000 array size

	Write operation	Read operation
Serial Solution	0.0005112000	0.0002246000
MPI Solution	0.0004411000	0.0004102000
Percentage Reduction	13.7128325509	-82.6357969724

Table of running time for settings 4: 4 processes, 1 000 000 array size

	Write operation	Read operation
Serial Solution	0.0054186000	0.0026625000
MPI Solution	0.0021251000	0.0032187000
Percentage Reduction	60.7813826450	-20.8901408451

Table of running time for settings 5: 4 processes, 10 000 000 array size

	Write operation	Read operation
Serial Solution	0.0667491000	0.0289420000
MPI Solution	0.0163440000	0.0253963000
Percentage Reduction	75.5142765970	12.2510538318

Table of running time for settings 6: 4 processes, 100 000 000 array size

	Write operation	Read operation
Serial Solution	0.6620104000	0.2823512000
MPI Solution	0.1749477000	0.2476461000
Percentage Reduction	73.5732701480	12.2914653807

Part D: Speed and Time Analysis:

Analysis for sorting algorithms:

- At 4 processes and 2 threads:
    + Merge and Quick sort MPI version both have a hard time competing with its serial version in run time
    + However, the MPI versions exceed in speed at 10 000 array length
    + Then, the larger the array size the closer the percent reduction gets to around 75% for Merge sort and around 69% for Quick Sort.
    + Enumeration sort MPI version works as expected with percent reduction around 95% for all array sizes.

- Explanation:
    + Compare to when running serial and OMP parallel version of Merge and Quick sort, the MPI-OMP versions runs much faster as they exceed the serial versions after second run at 10 000 array length. This may due to the fact that the initialisation of OMP threads are limited and MPI are already initialised at compile time.

+ However, the MPI-OMP version still slower to serial version at small array length because of the merging arrays from different processes as the time for sending and receiving time between different processes, although minimal, becomes more apparent the smaller the sorting time/size

Analysis for IO operations:

- At 4 processes:
    + The Write MPI version exceeds its serial version at the Third run with 100 000 array size
    + The Read MPI version exceeds its serial version at the Fifth run with 10 millions array size
    + The larger the array size the closer the percent reduction gets to around 70% for Write MPI and around 12% for Read MPI

- Explanation:
    + The Write MPI comparison with Serial Write is quite similar in performance to the Merge and Quick sort MPI version comparison. This most likely because the writing from array to file are simple and contains no overheads to slow the performance down. However, MPI Write still slower than Serial Write could be because of the MPI_Write_at function due to the seeking of the correct place first before writing can slow the algorithm down when the array size is small, making the seeking work time cost becomes apparent.
    + The Read MPI performs poorly compare to the Serial Read. At only 12 percent reduction, the performance improvement are negligible. This could be because although the Read from file work itself may be fast, the array itself are fragmented between different processes. In order to get the full array, the algorithm needs to perform MPI_Gatherv function. This slows the performance down dramatically from sending, receiving, and seeking the correct index in the root array before placing the sub arrays in.