ECEN 5033 Concurrent Programming

**LAB1 WRITE UP**

# **Introduction:**

This document is about the bucket sort and merge sort algorithms which are used to sort an unsorted text file and send it to a new or existing file in sorted manner.

**# Description of code organization:**

Consists of main,c and Makefile

* The lab1 code has been organized in two files which is the main.c and Makefile. It generates the executable mysort by running make.

**# Description of every file**

* Makefile compiles the code main.c and generates executable that can be run by typing make.
* main.c contains the main code of the project. This file takes the inputs as - - name and - - alg=fjmerge or - - alg=lkbucket. --name argument prints the full name of the person and alg argument takes options as fjmerge and lkbucket which sort the file according to the option specified.
* -t option takes the input as number of threads which needs to be run in parallel and execute the sorting operation
* It also takes a non-option argument which is the unsortedfile.txt. This file will have multiple lines which each line having integer of size 0 till INT\_MAX..
* It also takes -o argument which takes option as the sorted file eg.sortedfile.txt which will be created and contain the numbers in sorted manner for the unsorted file, in this case unsortedfile.txt. The numbers will be sorted and arranged line by line in the sorted file, in this case sortedfile.txt.
* The main.c contains the merge sort algorithm functions, bucket sort algorithm function and function to read and write to the file. Reading the file is done by using char \*\*array and converting the array to an integer 1d array. This array will be passed to the mergesort and bucketsort functions depending on the request from the user.
* The sorted array will then be written to a new or an existing file line by line.
* Merge sort is an out of place sorting algorithm which needs to create duplicate arrays to execute sorting. The time complexity of merge sort is O(nlogn) and in worst case is O(n^2).
* Bucket sort has O(n + k) time complexity for best case and O(n^2) complexity for the worst case. The space complexity for bucket sort is O(nk) for worst case
* The file is read using getline which takes row of the file as row of the array and columns of the file as columns of the array. This read array is converted using strtol to an integer array. The char \*\*array which reads input of from the unsorted file uses dynamic memory allocation and reallocates for file content greater than size of initially defined array.

# **Parallelization Strategy for Merge sort**

* The input file copied into an array will be divided into sub arrays where each individual thread will work on approximately same number of array elements.
* The low array index, high array index, array to be populated, thread number is calculated and passed for each thread by using a structure of these values
* New threads are created, and their specific indexes and unique parameters are passed to the merge thread function which will sort individual sub arrays using recursive merge sort.
* The threads are joined on completion of their unique tasks.
* The individual sub arrays are then finally merged to the final main array and is sent to the sorted\_file.txt

# **Parallelization Strategy for Bucket sort**

* The input file copied into an array will be divided into sub arrays where each individual thread will work on approximately same number of array elements.
* The number of buckets will be equal to number of threads.
* The low array index, high array index, array to be populated, thread number, individual array length and bucket divider is calculated and passed for each thread by using a structure of these values.
* Divider is calculated by dividing the max element of array by number of threads used to filter each element of array into separate sorted buckets, created using stl::map.
* New threads are created, and their specific indexes and unique parameters are passed to the bucket sort thread function where each thread elements are checked to see which bucket they will go to according to their values in comparison to value of max element in the total array. They are then added to individual sorted buckets.
* The threads are joined on completion of their unique tasks
* The individual sub arrays are then finally combined to the final main array to create a list of elements present in a sorted order.
* This sorted array is then written into a new or an existing text file.

**# Compilation instructions**

* Eg of how to run the project. First compile the file using **make** which creates an executable of the name **mysort**.

**# Execution instructions**

* Then run the executable using the command by typing

**./mysort - - name unsortedfile.txt -o sortedfile.txt -t10 alg=lkbucket**  which sorts the unsorted file using bucket sort taking 10 number of threads OR **./mysort - - name unsortedfile.txt -o sortedfile.txt alg=fjmerge** which sorts the unsorted file using merge sort and takes 10 number of parallel threads to execute the code.

**# Extant bugs**

* The input values are less than **INT\_MAX,** will lead to differences in the actual sorted file run using **sort -n** linux command and the **mysort** executable. Hence, the same should be considered when running the file.
* The datatype features of std::map throws erroneous results when taking very very large inputs leading to segmentaion fault.
* The std::map vector does not sort repeated numbers.