COL362 Assignment 3 Report
Efficient External K-way Merge-sorting

- Shreya Arora (2019CS10401)
- Eklavya Agarwal (2022VST9017)

Implementation and Design Choices

In this assignment, we have implemented a fast and efficient **K-way merge-sort** algorithm of a large number of character strings consisting only of ASCII characters. We have defined primarily 3 functions for our implementation. These are detailed as follows:

1. **createInitialRuns**

   This function takes 2 arguments:
   1) Constant reference to the string inputF that represents the file containing the input strings to be sorted
   2) Long integer key_count that represents the number of keys in the input file that need to be sorted.

   The function opens the input file as an ifstream object and initializes the variable 'runNum' that represents the number of initial runs created so far and the variable 'keys' to represent the number keys read from the input file so far. We then start reading lines from the input file till we reach the end of the file or till the keys read so far become equal to the key_count specified before.

   We create a temporary deque, 'temp' and keep on adding the read lines to it until the size of temp in bytes becomes equal to the MAXSIZE variable. We have defined MAXSIZE as 800 MB to ensure a more efficient implementation so that the number of passes can be minimized. This keeps on repeating until we reach the end of the input file or the keys variable becomes equal to key_count.

   After this, we sort the temp deque using the inbuilt sort function in the standard library and write the sorted output for the current run to the respective output initial run file whose name is defined according to the format temp.0.x where x is the runNum value increased by 1. Finally, we open this output file as an ofstream object and add its name to the list called 'runs'.

Design Choices for Ensuring Efficiency - createInitialRuns

We have used the deque data structure in order to leverage its efficiency in performing different functions like inserting and deleting elements at various indices as compared to a list or a vector.

We have used move semantics in order to prevent repeated copying of data in memory. This ensures a more efficient way of adding the read lines to the vectors/deques/lists used in the program.

In order to minimize the number of passes from the initial runs to the final output (where each pass involves merging of the created runs), we have defined a large value of the MAXSIZE variable so that the initial runs consisting of sorted data are as large as possible which subsequently results in a reducing the number of passes and eventually the overall wall clock time of the program.

Lastly, in order to ensure efficient memory accesses, we have used pointers and constant references of different memory objects in order to prevent multiple copies of the same objects in memory thus ensuring a more efficient memory management in the program.

2. **mergeRuns**

This function takes 2 arguments:

1. Constant reference to the list inputFiles that represents the list containing the initial run files to be merged
2. Constant reference to the string outputFile that represents the output file to be updated on merging the 2 input files

In this function, we merge the input files in a sorted order into the output file using a priority queue for the sorting process. We begin by opening the input files and the output file and creating a priority queue pq consisting of Node objects, struct that contains a line of text and the index of the file that the line came from. We have ordered the priority queue pq based on the text line to ensure that the smallest line is at the top of the priority queue.

Then, we read the first line from each of the input files, create a Node object for each of those read lines and insert them into the priority queue pq. We then enter the loop and keep on removing the topmost or the smallest element of the priority queue and writing it to the output file, subsequently reading the nextline from the respective file (whose line was added to the output file) and adding it to the priority queue. In case we have reached the end of this file then, we move on to the next element in the priority queue. Finally, we close all the input files and the output file.

<u>Design Choices for Ensuring Efficiency - mergeRuns</u>

We have used the priority queue data structure to find the minimum element in each iteration of the loop as a priority queue always maintains the smallest element in it as the topmost element and thus finding the smallest element in the priority queue while merging the different runs becomes very efficient and helps in minimizing the overall time required for merging the different runs created. This is particularly helpful when the datasets being processed are very large.

We have used move semantics in order to prevent repeated copying of data in memory. This ensures a more efficient way of adding the read lines to the vectors/deques/lists used in the program.

Lastly, in order to ensure efficient memory accesses, we have used pointers and constant references of different memory objects in order to prevent multiple copies of the same objects in memory thus ensuring a more efficient memory management in the program.

3. **external_merge_sort_withstop**

This function has the exact signature as specified in the problem statement. It takes he following arguments:

1) *input:* Full path of the file containing input strings
2) *output:* Full path of the file where the sorted output should be written
3) *key_count:* Number of keys in the input file that need to be sorted.
4) *k*: the maximum fanout of the merges that are allowed. Default value = 2.
5) *num_merges:* Optional argument given to stop the sorting process midway and inspect intermediate results. If not specified (or set as 0), function continues until the final sorted output is produced. Otherwise, it stops after performing num_merges number of merge steps, where a merge-step is defined as the reading (at most) k runs, and writing out a merged output to a file.

In this function we implement the entire external K-way merge sort algorithm with a stopping condition based on the number of merges. We begin by creating the initial runs by calling the createInitialRuns() function defined above. After this, we initialize the stage as 1 signifying the initial runs stage. Then, we check the value of the num_merges variable specified and in case of a 0 value, we allow the function to run till it completely sorts the input file by assigning it a MAX value. We initialize the 'totalMergesSoFar' variable which is maintained to check for the stopping condition for external sort.

We check for the number of initial runs created. In case, this value is 1, then we simply update the output file with the contents of the run file created, close the input and output file and return 0 as no merges were performed in this scenario. In case more than one run is created, we enter a loop that continues until only one run is left. In each iteration, we merge k runs at a time, store the merged runs output in a temporary file, and add the temporary file to the list of runs in order to merge it later in the next iteration of this loop.

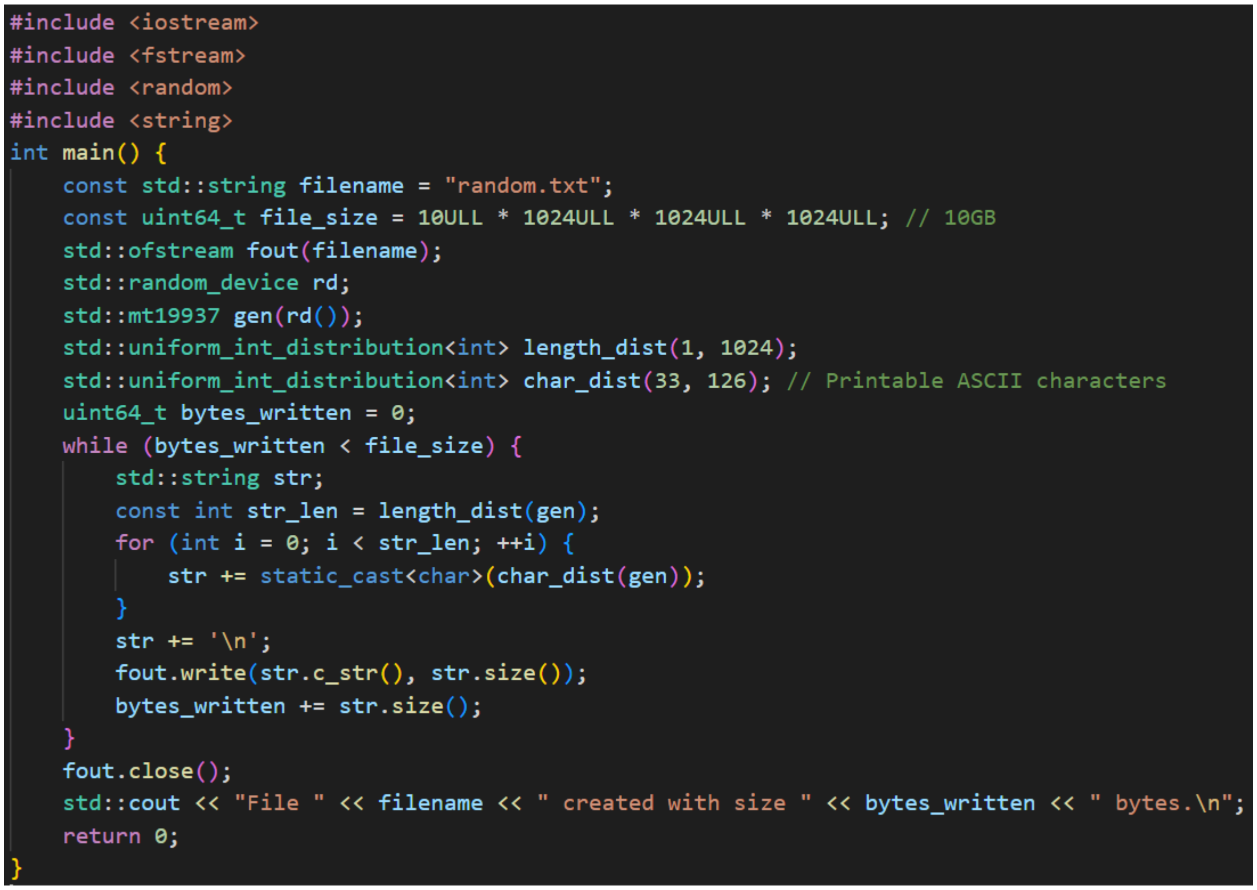
We keep on updating the 'totalMergesSoFar' variable in each iteration of the loop and stop when the number of merges exceeds num_merges. Finally, when only one run is left at the end of the loop, we simply update the output file with the contents of the run file, close the input and output files and return the number of merges performed so far to indicate successful completion of the function. In case of any errors, we return -1 to indicate an incomplete completion of the external sort algorithm.

Performance Evaluation Methods

In order to evaluate the performance of our algorithm, we built and executed it on a Windows system with the following specifications:

- Processor: Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz   2.90 GHz
- Installed RAM:16.0 GB (15.9 GB usable)

We have used the -O3 compiler optimization flag, with no debug symbols enabled for the testing and evaluation process. In order to test our algorithm with datasets of different sizes and optimize the wall clock time of the program to complete the sorting of the input file, we created a testcase.cpp file to create files with variable sizes consisting of strings with variable lengths. The function used for this process is as follows:



```cpp
#include <iostream>
#include <fstream>
#include <random>
#include <string>
int main() {
    const std::string filename = "random.txt";
    const uint64_t file_size = 10ULL * 1024ULL * 1024ULL * 1024ULL; // 10GB
    std::ofstream fout(filename);
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> length_dist(1, 1024);
    std::uniform_int_distribution<int> char_dist(33, 126); // Printable ASCII characters
    uint64_t bytes_written = 0;
    while (bytes_written < file_size) {
        std::string str;
        const int str_len = length_dist(gen);
        for (int i = 0; i < str_len; ++i) {
            str += static_cast<char>(char_dist(gen));
        }
        str += '\n';
        fout.write(str.c_str(), str.size());
        bytes_written += str.size();
    }
    fout.close();
    std::cout << "File " << filename << " created with size " << bytes_written << " bytes.\n";
    return 0;
}
```

Function to create a random text file of size 10 GB

In order to measure the wall-clock time used by our program to complete the sorting in a single-threaded program, we used the chrono library and counted the number of milliseconds elapsed between the start and end of the function call. For each workload, we ran our program 3 times with 3 different values of k (= 2, 8, 16), and evaluated the average of all runs as the measure of the program performance. The analysis of the performance measure results obtained on different workloads is detailed in the next section.

Performance Analysis

Test Size	k = 2	k = 8	k = 16	Avg Performance
9.56 MB	0.36	0.23	0.17	0.25
20.5 MB	1.77	2.00	1.79	1.85
47.8 MB	1.41	1.75	1.47	1.54
95.6 MB	2.45	2.33	2.78	2.52
478 MB	17.38	16.14	10.92	14.81
488 MB	12.21	12.19	12.57	12.32
956 MB	42.13	43.90	39.15	41.72
4.67 GB	607.70	287.68	247.46	380.95
9.33 GB	626.81	298.54	278.51	401.29
10 GB	634.53	300.66	274.01	403.07
20 GB	1,289.07	645.31	532.06	822.15

We created test cases with sizes as shown in the above table in order to evaluate the performance measure of our algorithm. For each workload that we have used, we ran the program 3 times with 3 different values of k (= 2, 8, 16), and then evaluated the average of all runs as the measure of our program performance. The highlighted rows in this table signify the performance of our program over the given datasets. The 20.5 MB case is the *english-subset.txt* workload and the 488 MB case is the *random.txt* workload.