Implementing External k-way Merge-Sort

COL362 Assignment 3

Nischay Diwan
2020CS50433

Dakshit Babbar
2020EE30163

April 2023

1 Objective

Given an input file with a string of ASCII characters in each line, we are supposed to sort these strings in increasing order, for which we have to implement the External k-way Merge-Sort algorithm. This algorithm is used to sort the input when the memory size available for sorting is significantly less than the input size itself. We are given the following specifications:

- The memory size to be used is 1GB.
- The maximum fanout of each merge is `k`
- The number of keys to be sorted is `key_count`
- The number of merges to be performed is `num_merges`

2 The Algorithm

The External k-way Merge-Sort Algorithm will sort a given input file of large size which is not able to fit in the memory completely. The algorithm runs in the following two steps:

1. **Sorting** Load the contents of the input file into memory, fitting in as much as possible. Sort this smaller input present in the memory and create a smaller file to store these locally. We call this smaller sorted file, one "run". Make many smaller runs out of the bigger input file.

2. **Merging** Take at most `k` runs together and merge them in the following way. Load first few elements of each of these `k` runs in the memory making sure that elements from each run receive equal space in the memory and some space remains for the output buffer.

   Find the minimum e_m of the elements present in the memory and transfer it to the output buffer and replace the e_m with the next element present in the run to which e_m belonged to. Do this until all the runs get empty and keep dumping the output buffer, once it gets filled, to a new file locally.

   Perform the above merge operation until one run is left. Transfer the contents of this run to the output file.

3 Sorting

We implemented in-memory sorting using the min-heap data structure. The contents loaded from the disk into the memory is converted into a min-heap data structure. We then use the heap-sort algorithm for the sorting of these elements.

We preferred the use of heap-sort over merge-sort and quick-sort because it uses $O(1)$ auxiliary space whereas the other two use $O(n)$ auxiliary space even though all three have a time complexity of $O(nlogn)$. This speeds up our sorting step and also does not use the limited memory that we have for the sorting of the elements.

An important feature of our implementation for sorting is that to use the memory most efficiently, we count the number of characters each string has, as soon as the number of characters equals the bytes that the memory can store we stop getting any more strings from the disk and process the elements presently in the memory.

4 Merging

For the merging step, we divide the memory(= 820MB) equally between the `k` runs. Also the output buffer gets the remainder of this division and extra 100MB. Now to efficiently find the minimum element e_m among the elements present in the memory we use the min-heap data structure. With this implementation we are able to retrieve e_m in $O(logn)$ time where n is the number of elements present in the heap. We will also be able to insert the new element (that replaces e_m) in $O(logn)$ time. Hence we get an $O(nklogn)$ time merging of the k runs.

Note that this implementation is better than the simple array implementation where we need to traverse the array every time to retrieve e_m and hence takes $O(n)$ time even though we get an $O(1)$ insertion of the new element which gives an $O(n^2k)$ merging of the k runs.

An important feature of our implementation for merging is to open streams of all the `k` run files at once, read them till the algorithm ends and then close them all together. This is a better approach than to open the file just when we want and then close immediately. Which will require us to open the file again if we want to fetch the next element from it, which will be inefficient because we will have to traverse till the offset every time we open the file.

5 Performance Evaluation

We learnt that the time taken by our implementation depends on the size of the data-set as well as the memory size that we consider. We verify the correctness of our implementation by using some self prepared data-sets as described below. We consider the wall clock time for a given memory size as the performance measure.

We prepared a small data-set (600B) using random words in the English language for the purpose of debugging

Memory Size	**k=2**	**k=8**	**k=16**
50B	3ms	1ms	0ms
100B	1ms	0.5ms	0.01ms
500B	1.1ms	0.8ms	0.023ms

We then prepared a decently large data-set (1.1GB) using random 512 characters(alpha numeric) long strings totalling to 2.1 x 10^6 strings for the verification purposes

Memory Size	**k=2**	**k=8**	**k=16**
100MB	66.64s	48.60s	42.79s
500MB	40.4s	38.71s	28.95s
800MB	38.2s	40.1s	37.6s

We then prepared a larger data-set (4GB) using using random 1024 characters(alpha numeric) long strings totalling to 4 x 10^6 strings for the verification purposes

Memory Size	**k=2**	**k=8**	**k=16**
100MB	12.12mins	6.75mins	6.14mins
500MB	8.78mins	4.15mins	3.92mins
800MB	4.01mins	2.23mins	76secs

6 Testing

Performance on the `english-subset.txt` data-set:

Memory Size	**k=2**	**k=8**	**k=16**
1MB	15.34s	10.21s	7.64s
10MB	7.97s	5.39s	5.4s
800MB	2.8s	3.0s	2.7s

Performance on the `random.txt` data-set:

Memory Size	**k=2**	**k=8**	**k=16**
25MB	43.5s	26.1s	22.09s
100MB	16.67s	13.56s	12.98s
800MB	5.39s	6.12s	4.93s

7 Notes

1. We consider that total memory for the program is limited for 1GB. Subsequently, we take total memory into our usage of 820MB. There is also some extra memory of 100Mb allocated to the output buffer while merging.
2. We return -1 from the function when input file does not exists or unable to open it. Else it always returns the number of merges it has completed.
3. After all the final file is generated, we rename it to required output file.
4. It is recommended that previous temp file need to be removed for better efficiency.
5. When the num_merges is non-zero and less than or equal to the expected number of merges then we do not generate the output file and just complete all the merges and make all the necessary temp files.