COL362 Assignment 3

Ankit Raushan (2020CS10324)
Sibasish Rout (2020CS10386)

Design choices

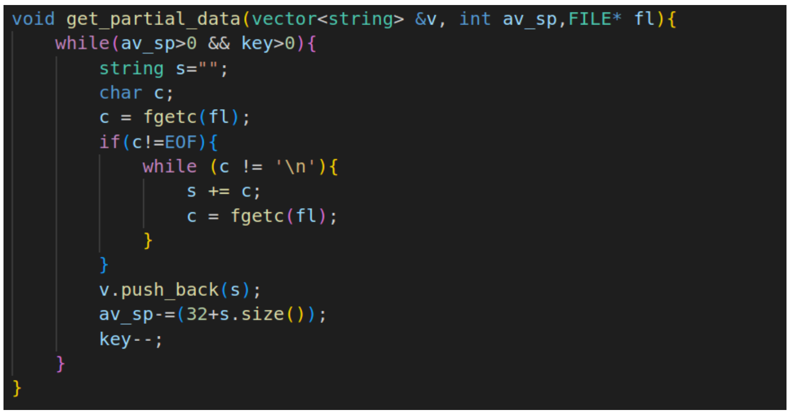
i) Algorithm

The most common sort-merge approach used in external merge sort is a hybrid one. A temporary file is read, sorted, and written out from data chunks that are tiny enough to fit in main memory during the sorting process. The sorted subfiles are concatenated into a single, bigger file during the merge stage.In this way if the implementation is memory friendly the memory usage for sorting can be reduced by a large amount. Even we can introduce memory usage restrictions and still get the result in sufficient time.

ii) Implementation

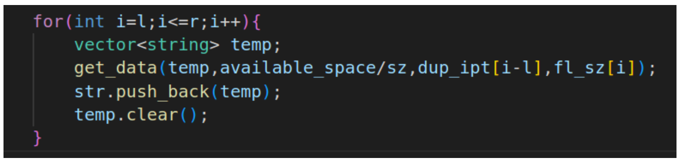
The implementation steps are as follows

a)In the first step we create runs from the input file based on the memory usage restriction. For this step we use the fgetc function which reads 1 character at a time and also stores a pointer to the last character read. Once a run is read and compiled into a vector of strings it can be sorted using the inbuilt sort function and then stored into temporary text files which can then be read later to produce the combined output. We then also cleared the temporary vector after writing it in a file. This creates temp.0.* files according to the memory available.



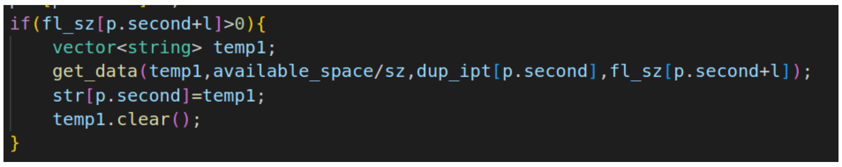
```cpp
void get_partial_data(vector<string> &v, int av_sp,FILE* fl){
    while(av_sp>0 && key>0){
        string s="";
        char c;
        c = fgetc(fl);
        if(c!=EOF){
            while (c != '\n'){
                s += c;
                c = fgetc(fl);
            }
        }
        v.push_back(s);
        av_sp-=(32+s.size());
        key--;
    }
}
```

b)The second step is the most important stage which is of merging k files. We merge k files at a time which is passed as an argument to the external_merge_withstop() function. When we merge k files we bring available_space/k size chunk from each file then merge it .



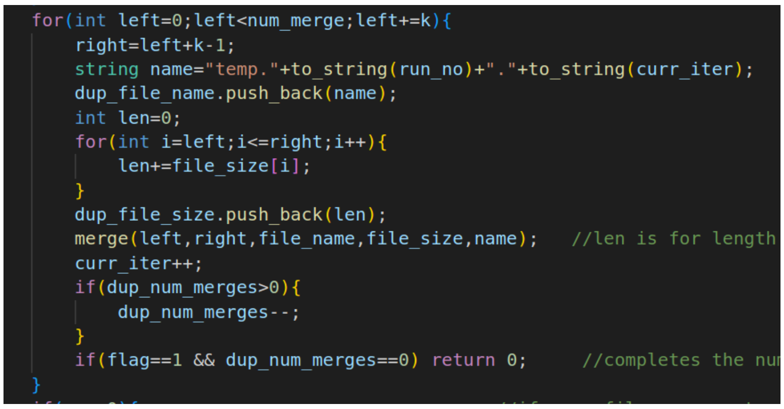
```cpp
for(int i=l;i<=r;i++){
    vector<string> temp;
    get_data(temp,available_space/sz,dup_ipt[i-l],fl_sz[i]);
    str.push_back(temp);
    temp.clear();
}
```

When all the string of a chunk is used up in merging then we bring available_size/k size chunk from that file again so that the memory space is never exceeded

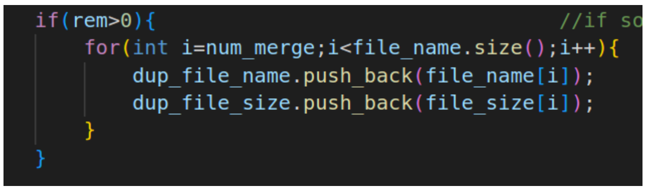

```
if(fl_sz[p.second+l]>0){
    vector<string> temp1;
    get_data(temp1,available_space/sz,dup_ipt[p.second],fl_sz[p.second+l]);
    str[p.second]=temp1;
    temp1.clear();
}
```

This completes merging k files

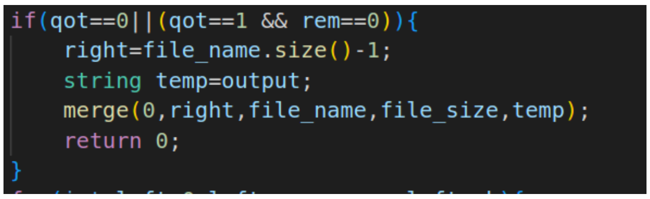
c) One run is generated from previous run by merging k files at one time. Then next k files from previous run are merged to generate other file in this run. This is all done with the help of a for loop


```
for(int left=0;left<num_merge;left+=k){
    right=left+k-1;
    string name="temp."+to_string(run_no)+"."+to_string(curr_iter);
    dup_file_name.push_back(name);
    int len=0;
    for(int i=left;i<=right;i++){
        len+=file_size[i];
    }
    dup_file_size.push_back(len);
    merge(left,right,file_name,file_size,name);   //len is for length
    curr_iter++;
    if(dup_num_merges>0){
        dup_num_merges--;
    }
    if(flag==1 && dup_num_merges==0) return 0;     //completes the nu
}
```

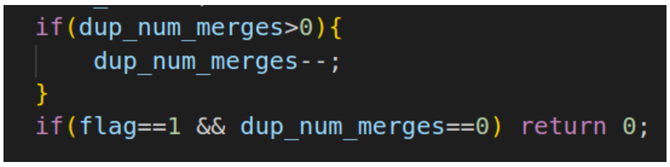
If there are some remaining files in the previous run whose number is less than k then they are not merged but appended in the present run to be merged later.


```
if(rem>0){                                    //if so
    for(int i=num_merge;i<file_name.size();i++){
        dup_file_name.push_back(file_name[i]);
        dup_file_size.push_back(file_size[i]);
    }
}
```

Finally output file is generated when in a run number of files become less than or equal to k.


```
if(qot==0||(qot==1 && rem==0)){
    right=file_name.size()-1;
    string temp=output;
    merge(0,right,file_name,file_size,temp);
    return 0;
}
```

d) If num_merges is greater than 0 then we have decreased num_merges by 1 every time a merge completes. Flag is used to check whether we have to terminate using num_merges or not


```
if(dup_num_merges>0){
    dup_num_merges--;
}
if(flag==1 && dup_num_merges==0) return 0;
```

This completes our implementation.

iii) Data-structures used

The important data structures used are
a)vector
The standard stl vector data structure is used to sore as well as read the runs at different stages of the merging phase.The inbuilt sort function is applied on the runs stored in form of vectors in the initial phase.
b)priority_queue
This is used to implement the heap data structure and to stored the lexicographically least value from each run as it performs in-memory sorting in only O(logn). The insertion and deletion are each O(logn) time complexity and this helps a lot in reducing the time taken for multiple sorting steps which would otherwise have to be done.

Optimisations

i) Algorithm Optimisations

Our implementation required multiple sorts at each step and required the structure to be sorted at each moment. Thus the use of heap gave us a huge improvement in the measuredd time values.Additionally we have used a balanced multiway merge which ensures that the number of disk I/O operations are less.Here we have used a hybrid algorithm where the smaller blocks are sorted using the in-built sort algorithm and the larger blocks are merged according to the external merge sort algorithm using a balanced multiway approach.

ii) I/O Optimisatiions

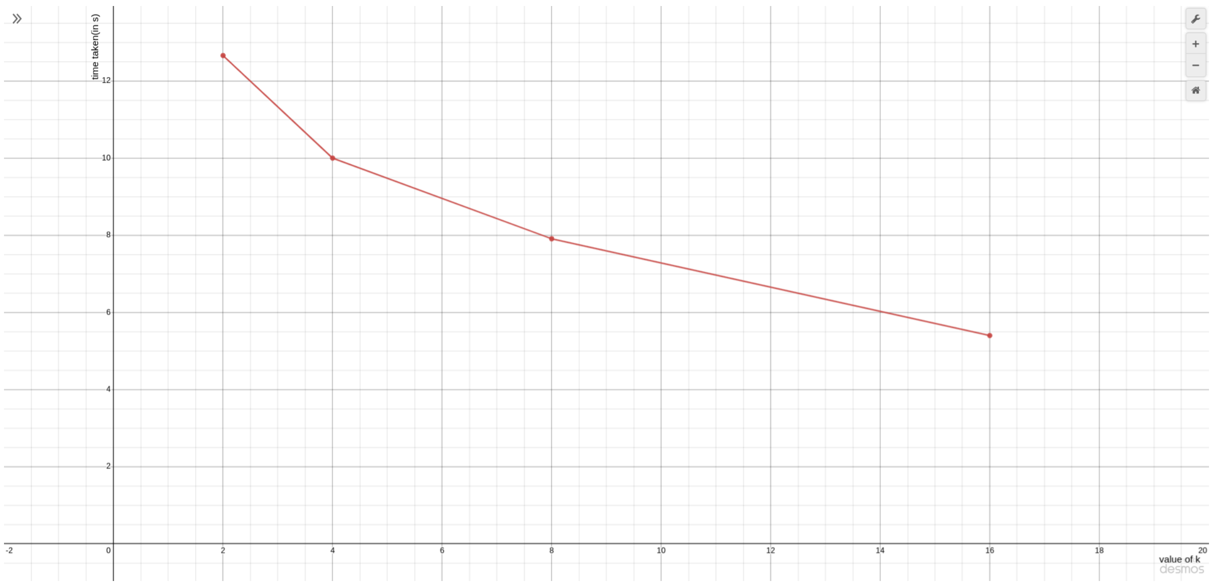
Since data must be read from and written to disc several times during the sorting process, external merge sorting performance is essentially constrained by the speed of I/O operations. Hence, optimising I/O operations is essential for raising external merge sorting performance.The optimisations we tried included block writing to the output stream where instead of writing the output to the file every time we store it in a bufferand write to the file only when the buffer size exceeds a certain limit. However this optimisation resulted in very limited speedup so we did not include it in our final submission.

Performance

For random.txt
Assuming maximum space of 50Mb

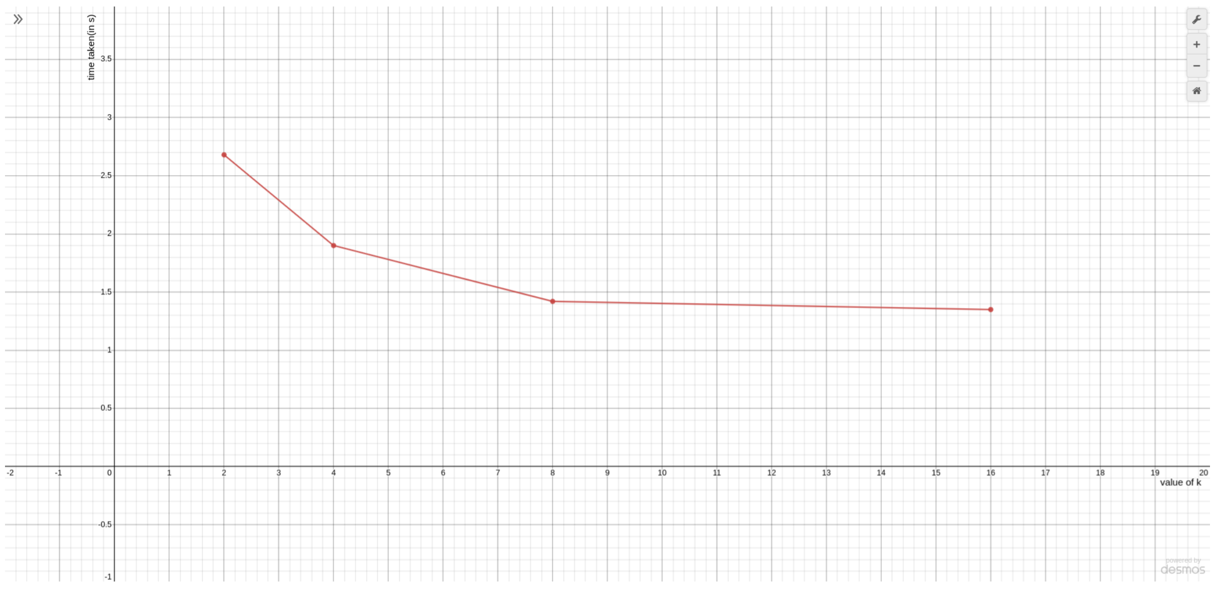
Value of k	**Time taken(in s)**	**Memory usage(in kbytes)**
2	12.66	109600
4	10.00	109612
8	7.91	109596

16	5.4	109616



For english-subset.txt
Assuming maximum space of 2Mb

Value of k	**Time taken(in s)**	**Memory usage(in kbytes)**
2	2.68	10416
4	1.90	9852
8	1.42	9868
16	1.35	9840



Observations

We can clearly see that as we are increasing k time taken in external k merge sort withstop decreases. This we have also seen theoretically in class in the time complexity of external k merge sort in which k appears in the base of logarithm. Hence increasing fanout decreases the time taken. This we can also see from the graph.