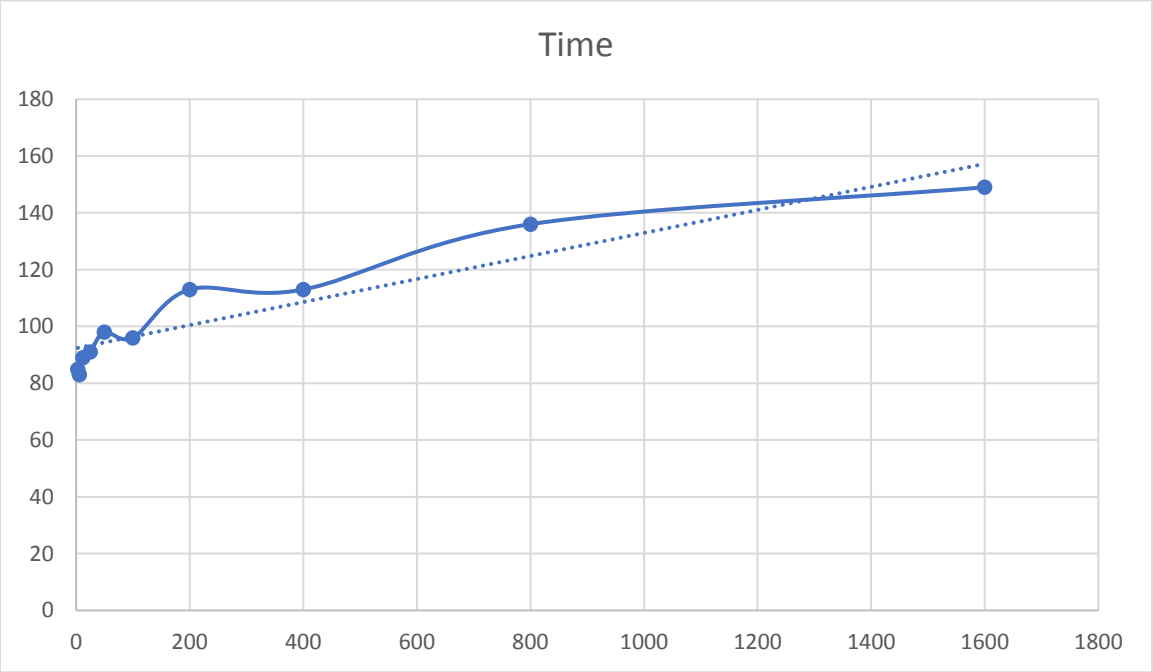
Graphs

This table shows the relationship between the increasing of mappers *2^n and the time:

Time	Mappers	Reducers	Sample size
85 Sec	3	100	1000
83 Sec	6	100	1000
89 Sec	12	100	1000
91 Sec	25	100	1000
98 Sec	50	100	1000
96 Sec	100	100	1000
113 Sec	200	100	1000
113 Sec	400	100	1000
136 Sec	800	100	1000
149 Sec	1600	100	1000

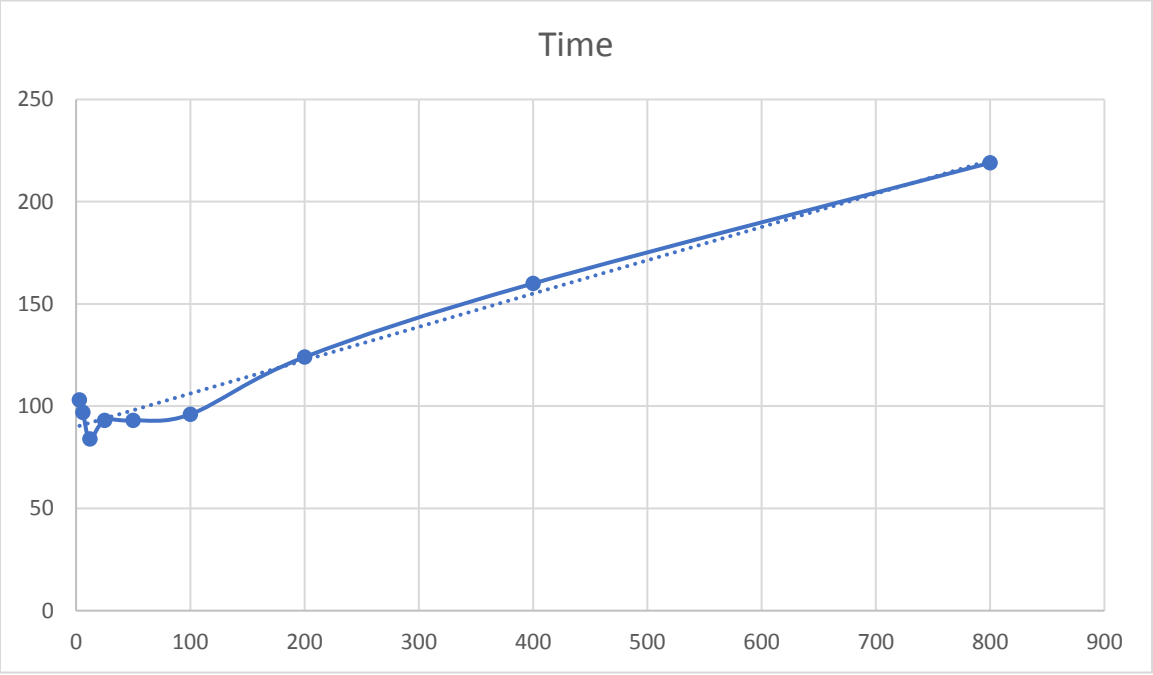
This graph represents the table:



This table shows the relationship between the increasing of reducers *2^n and the time:

Time	Mappers	Reducers	Sample size
103 Sec	100	3	1000
97 Sec	100	6	1000
84 Sec	100	12	1000
93 Sec	100	25	1000
93 Sec	100	30	1000
96 Sec	100	100	1000
124 Sec	100	200	1000
160 Sec	100	400	1000
219 Sec	100	800	1000

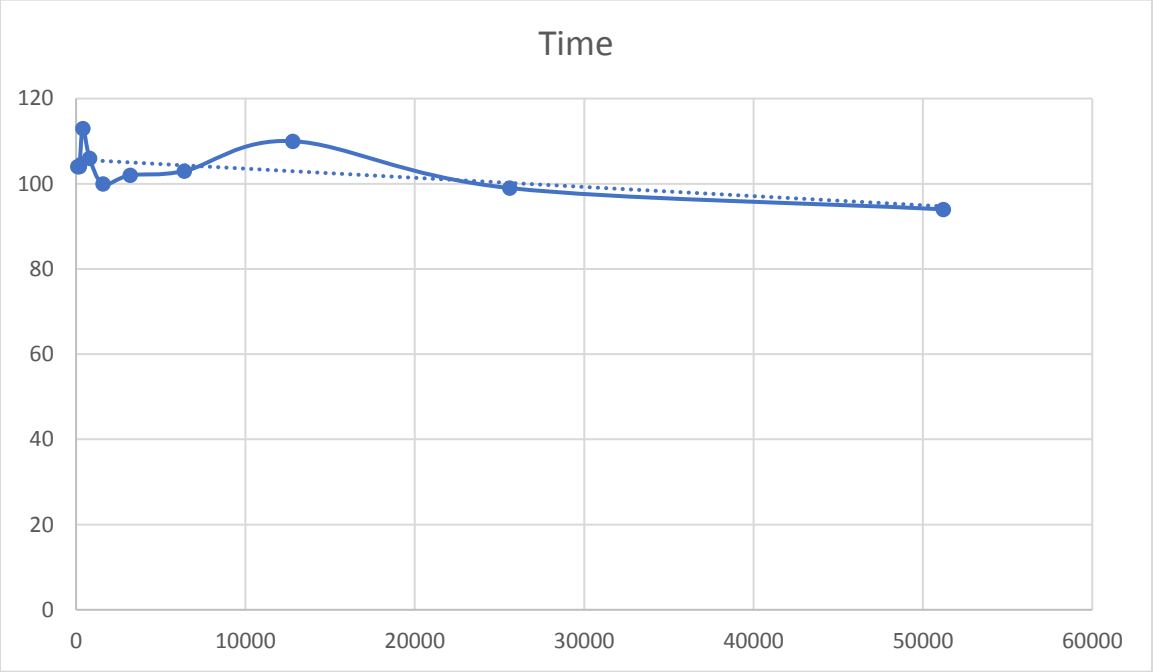
This graph represents the table:



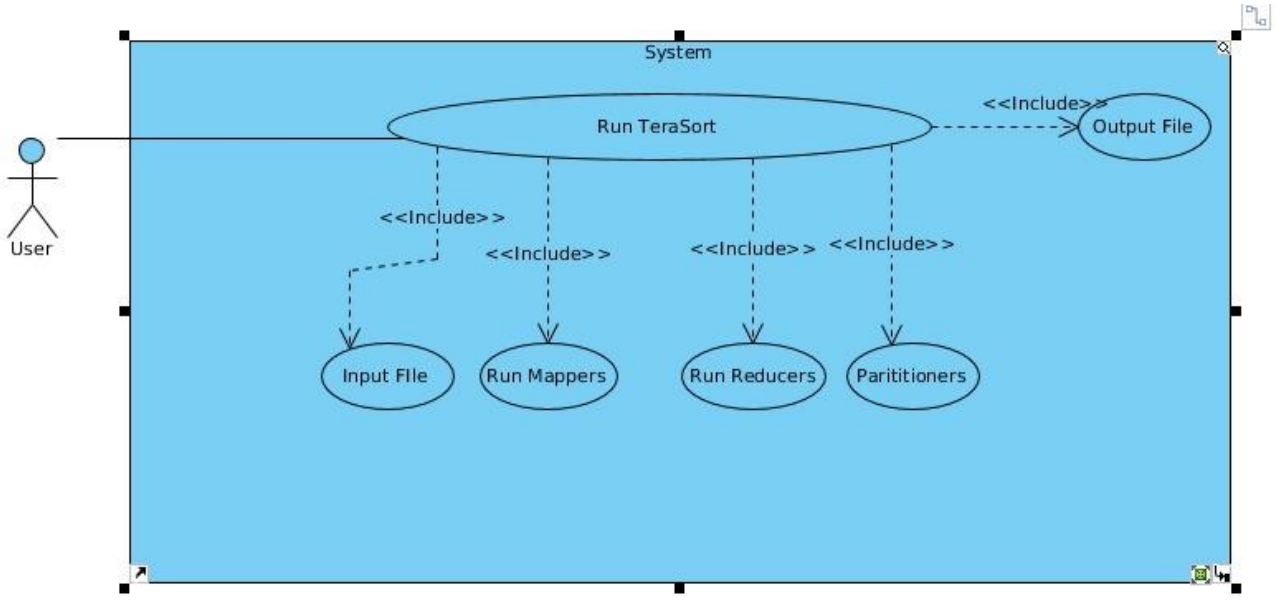
This table shows the relationship between the increasing of sample size *2^n and the time:

Time	Mappers	Reducers	Sample size
104 Sec	100	100	100
104 Sec	100	100	200
113 Sec	100	100	400
106 Sec	100	100	800
100 Sec	100	100	1600
102 Sec	100	100	3200
103 Sec	100	100	6400
110 Sec	100	100	12800
99 Sec	100	100	25600
94 Sec	100	100	51200

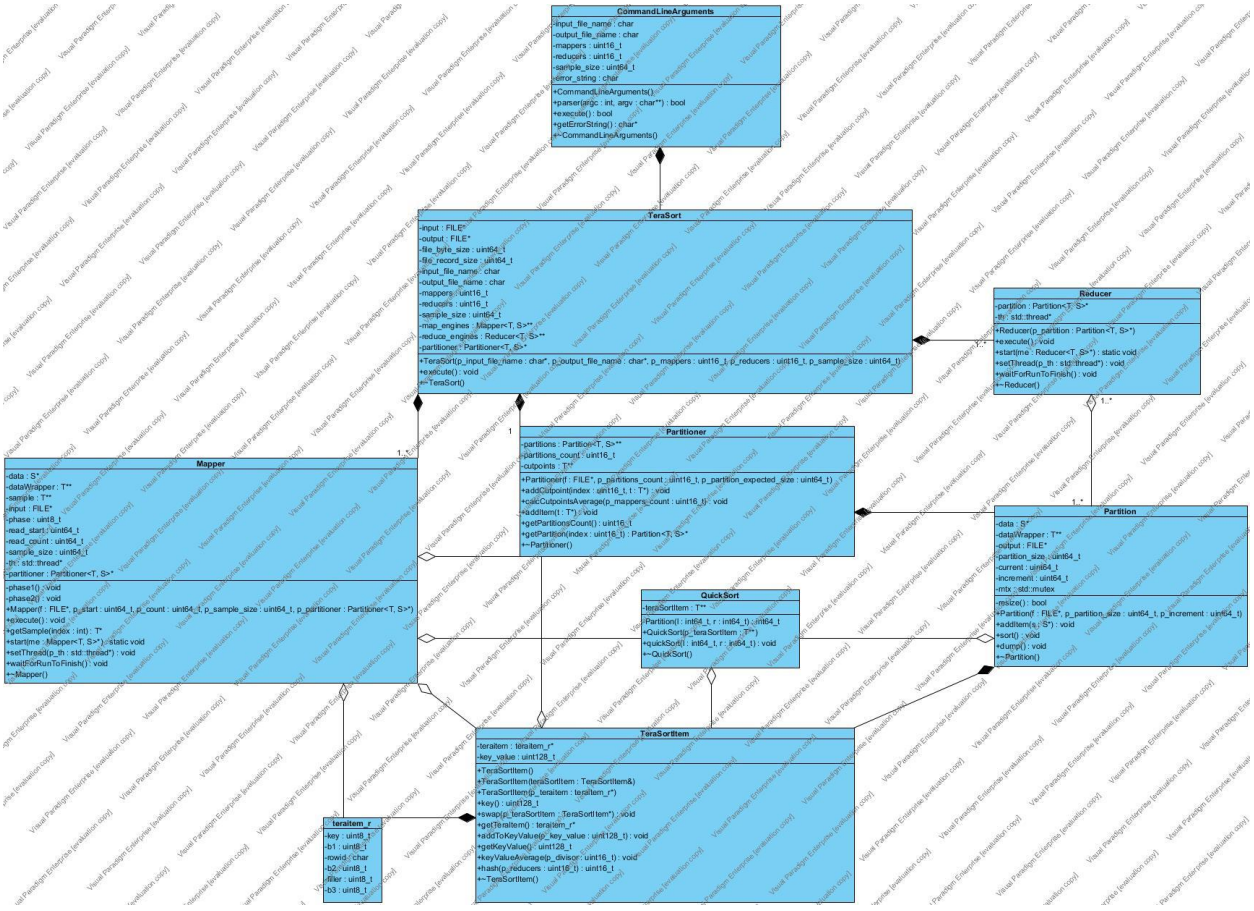
This graph represents the table:



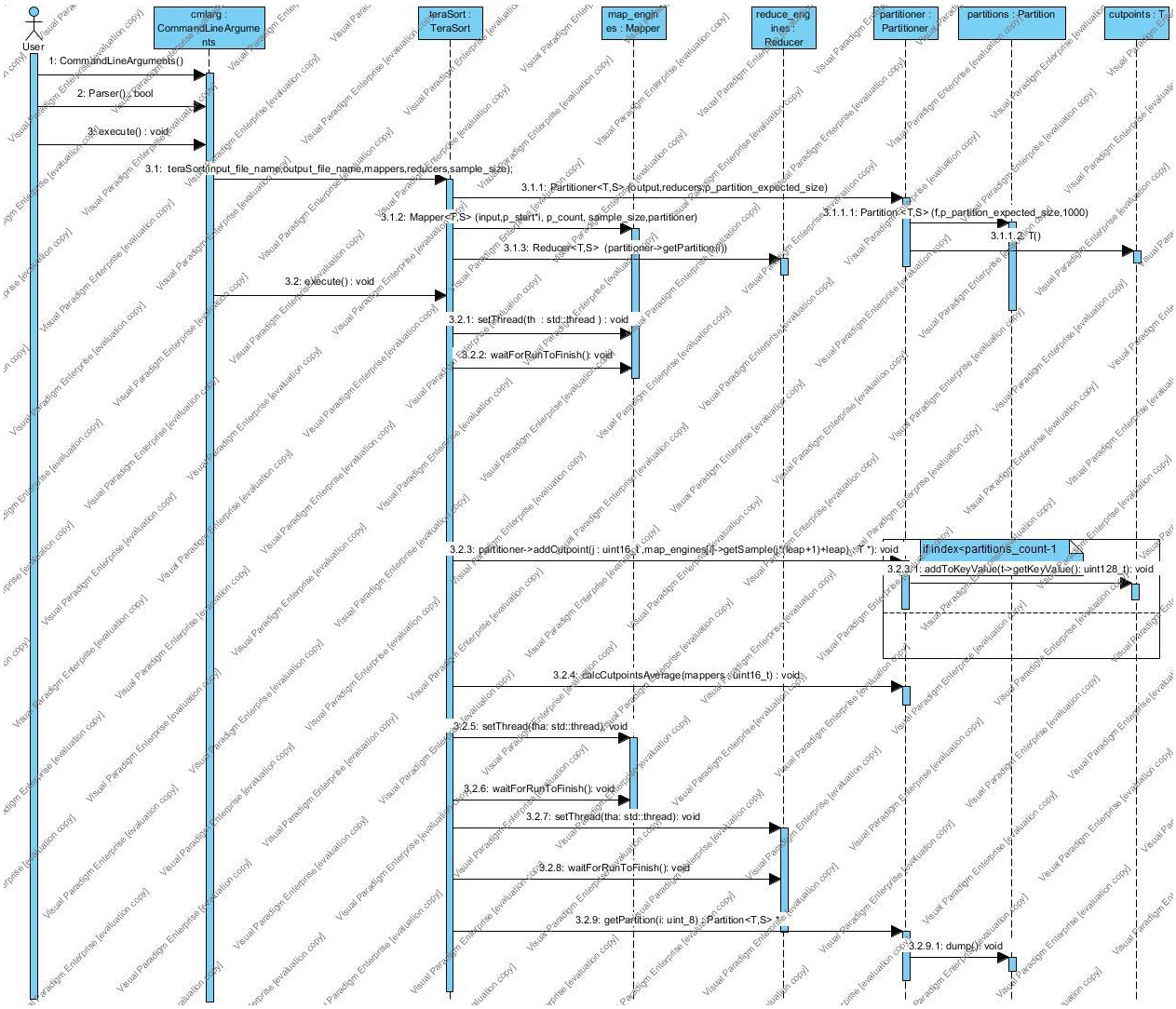
Use case diagram



Class diagram



Sequence diagram



Assumptions:

1- **In Mapper<T,S>::phase1() , we assumed that filling the cut points array in the partitioner class inside the phase1() will result in race condition between the multiple running threads. Thus, we concluded that we should either have a shared mutex around this function to make these processes atomic or even save each mapper cut points array in its object, then return it to terasort object of type Terasort class in Terasort.cpp. Eventually, we would end up with a loop that fills the cutpoints array in the partitioner without a racing condition.**

2- **The partition expected size should be (File_size/record_size)/number of reducers; however, this assumption was not true because not every partition should be filled with some date, maybe the cut points restrict some partitions from having any element filled in it. Thus, we filled the expected size parameter with zero in order to avoid "core dumped" error.**

3- **The larger the sample range the more accurate the cut points which results in perfectly organizationally sorted partitions.**

4- **Dividing the read file to N number of mappers will usually result in N-1 equal sizes (S1) of mappers and 1 mapper of size (S2) greater than (S1) the size the first N-1 mappers.**

5- **The size fo the first N-1 mappers in the mapper engine will be (S1) which will be equal to = (File_size/record_size)/number of mappers.**

6- **We assumed that using a reservoir sampling technique, which is used in real Hadoop environments, will protect us from picking up the same sample element by any mean. Thus, we did not use B1, B2, and B3 variables found in the teraitem_r struct.**

7- **Increasing the number of mappers and reducers would lead to the increase in the running time of the program.**

8- **Increasing the number of samples taken to make our cut points will, to a certain limit, decrease the running time of the program, because we would end up with perfectly organizationally sorted partitions which will decrease the time of the sorting and swapping and eventually will decrease the running time of the program, then it will increase it.**

Our design approach to solve this problem and why do you think it is the most optimum approach:

1- **We read the code starting from "parallel tera sort" file to understand the flow of sequence.**

2- **We started to write the big function calls to make the expected tasks**

3- **Then we wrote the small function calls to make the required tasks**

4- **After that, we wrote the implementation of the functions**

5- **We revised the includes to ensure the suitable connection between files**

6- **Follow the errors that showed**

7- **Finally test the code on the different sources**

The reason why our code the optimum one:

1- **We run our program on multiple threading guarded with mutexes in order to make it mutex safe by locking the critical section and then unlock it at the end**

2- **We use the cutpoints methodology in order to sort the data in organizationally sorted partitions**

3- **We use the divide and conquer strategy in order to make our program a logarithmic function which will result in lower increasing rates.**

Complexity analysis

We deuced that the most expensive process that affects the final running time of the program is the quick sort. Thus, we analyzed the big oh of the quick sort.