Cloud Computing CS553 Spring 20

Homework 5

Sort on Single Shared Memory Node

April 13, 2020

Team 11

Shared Memory Sort Design

As part of the shared memory sort, we have implemented the following logic :

We assume that the RAM of the system is at least 8GB, so if we are sorting a file, which is less than 8GB, we can sort that file directly using the in-memory sort, which in our case is **Quick Sort**. However if a user intends to sort this small file using multiple threads we divide this file in smaller chunks(within the memory), apply quick sort individually and then merge them together into a single file.

If data size goes beyond 8GB, we opt for External sort. The way this works is we divide the data file into smaller temp files based on the number of threads and sort them individually using Quick Sort.

Users can choose the number of threads to be used for faster processing.

We spawn n threads at a time so that all thread sorts total data of 8GB(Quick Sort) at a go(Restricting memory usage below 8gb and also improving overall speed leveraging the power of hardware threads which in our case are 48), and this is repeated based on total temp files size. For example, if we have 16 temp files for 16GB we have 8 threads then processing will happen as follows:
8 threads sorting 8GB(1GB each) at a go X 2 cycles.

We wait for the spawned thread to complete before spawning a new batch of 8 threads to prevent additional RAM usage.

After this process we have sorted temp files of 1GB each.

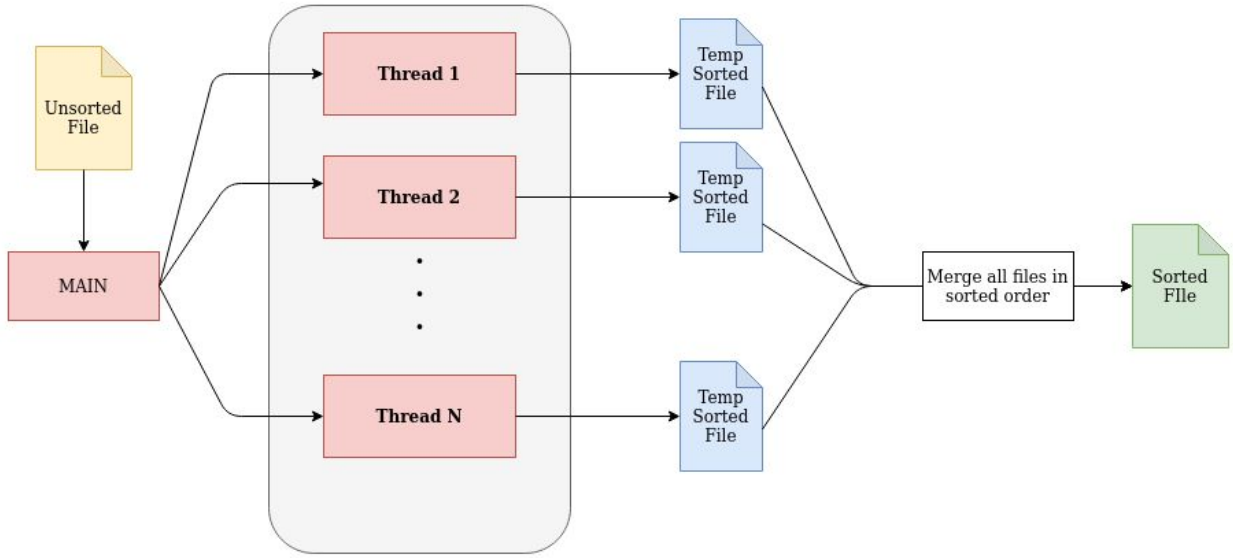

Fig 1. Code flow diagram

As we can see in the fig. Next step is to merge all this smaller temp files into a single file. We use the heap to do this. Initially, we put the 1st line from each file into **Min Heap** and pop the top element from the heap and write to the **final file**. We keep track of the file of which record/line is popped and put its next line into the heap.We repeat this process until our heap is empty, and thus we have a sorted file.

Design Tradeoffs

1) The viable number of threads is a tradeoff as we can only choose threads in the power of 2 i:e 1, 2, 4, 16 and 32. We can also choose 40 also as it perfectly divides the 8GB chunk and no loss of data is seen. If any number other than above mentioned is chosen, then there is possible data loss as the number won't divide the 8GB perfectly. This is done in order to keep the calculation of temp files as simple as possible.
2) Users cannot choose to restrict RAM size. As variable RAM size increases complexity within program calculation so as mentioned in the assignment we have fixed it to 8GB.

Problem :- The main problem is to sort the large data which can not fit the RAM. In these kinds of cases the external sort can be used to divide, sort and merge. First the data is divided into chunks and then chunks are sorted separately and finally merged to the main file.

Runtime Environment settings

NODE TYPE	compute_skyake
#CPUs	2
NUMBER OF THREADS	48
PROCESSOR MODEL	Intel Xeon @2.60Ghz
STORAGE VENDOR AND MODEL	Samsung (MZ 7KM240HMHQ 0D3)
RAM	192
OPERATING SYSTEM	Ubuntu 18.04

Results:

Exp	Shared Memory (1GB)	Linux Sort (1GB)	Shared Memory (4GB)	Linux Sort (4GB)	Shared Memory (16GB)	Linux Sort (16GB)	Shared Memory (64GB)	Linux Sort (64GB)
# threads	8	32	8	16	32	16	32	32
Sort Approach	In-memory	In-memory	In-memory	In-memory	External	External	External	External
Sort Algorithm	Quick Sort	Merge Sort	Quick Sort	Merge Sort	Quick Sort	Merge Sort	Quick Sort	Merge Sort
Data Read (GB)	1.01	0.98	3.94	3.9	15.65	15.62	62.52	62.49
Data Write (GB)	0.98	0.95	3.73	7.52	31.09	31.25	124.36	170.95
Sort Time (Sec)	14	10	54	51	192	219	818	1084
Overall I/O Throughput (MB/sec)	141.59	174.03	141.96	212.11	243.37	214.03	228.47	199.47
Overall CPU Utilization (%)	96.5	84	99.06	86.56	98.2	86.5	99.41	87.04
Avg Memory Utilization (GB)	1.84	1.67	8	5.49	2.16	7.44	2.32	7.83

Difference in performance:

As seen from the table above **MySort** performs at par with **Linux sort** (systems memory was restricted to 8 GB). However in some cases we tend to perform better and vice-versa. Following is the breakdown of the performance achieved by both the programs:

1GB Workload:

As the file is of less than 8 GB we use in-memory sort for this file.

The shared memory sort **(MySort)** is a bit slower as compared to Linux Sort. Our cpu usage is 96.5% percent and average memory utilization is also of 1.84 which is less than that of linux sort.

We achieved optimal performance using 8 threads while Linux sort achieved the optimal performance with 32 threads.

4GB Workload:

As the file is of less than 8 GB we use in-memory sort for this file.

Overall CPU and Memory utilization is higher in our case **(MySort)**. However, overall I/O throughput in case of Linux sort better. So, they tend to balance each other and we get similar performance.

We achieved optimal performance using 8 threads while Linux sort achieved the optimal performance with 16 threads.

16GB Workload:

As the file is of greater than 8 GB we use external sort for this file.

Shared memory sort **(MySort)** performs faster than Linux sort for two reasons: 1) Overall CPU utilization is ~100 2) overall I/O throughput is more than that of Linux sort.

We achieved optimal performance using 32 threads while Linux sort achieved the optimal performance with 16 threads.

64GB Workload:

As the file is of greater than 8 GB we use external sort for this file.

Shared memory sort **(MySort)** performs faster than Linux sort for two reasons: 1) Overall CPU utilization is ~100 2) overall I/O throughput is more than that of Linux sort.

We achieved optimal performance using 32 threads while Linux sort achieved the optimal performance with 32 threads.

Following are the plots for 64GB data sort.

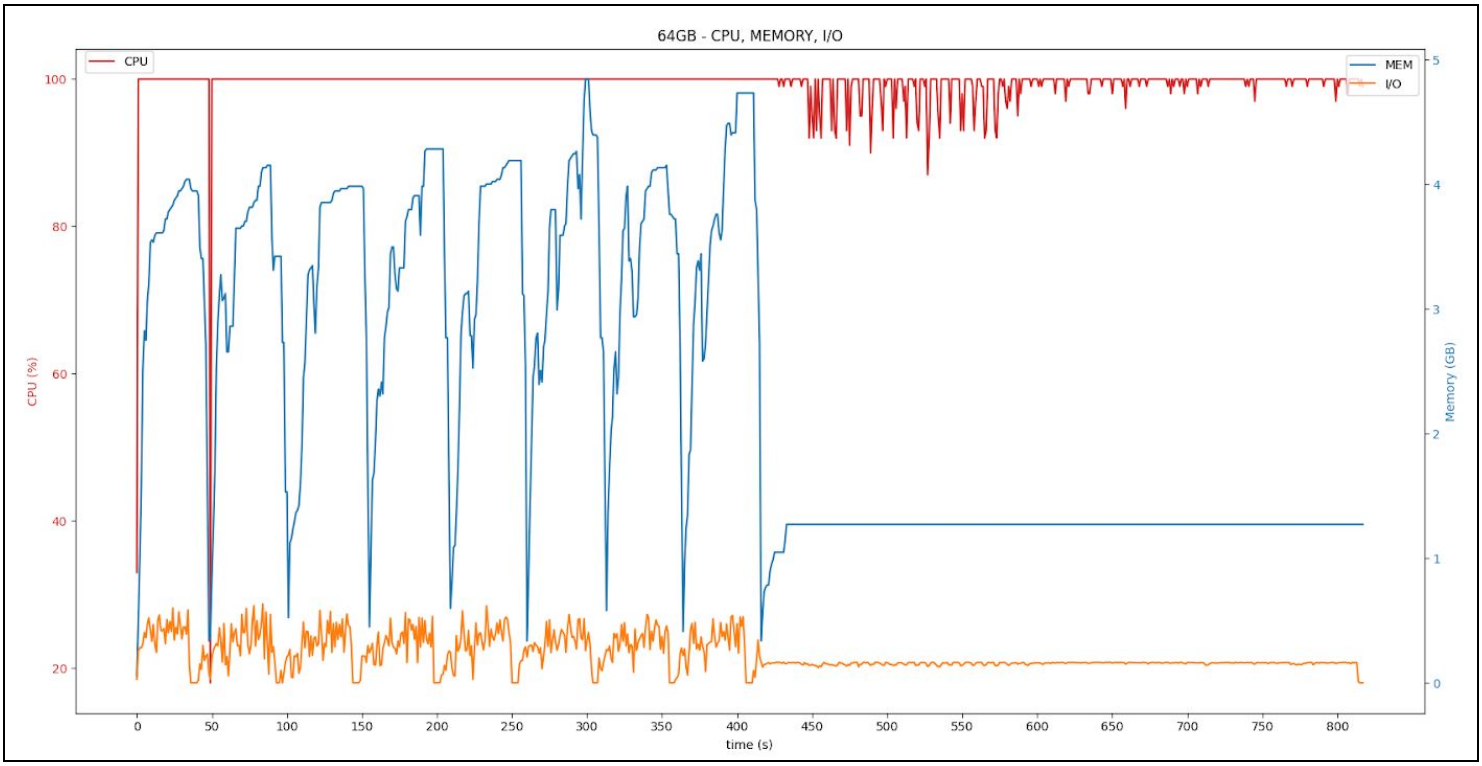

Figure 2. CPU vs I/O vs Memory utilization Shared Memory

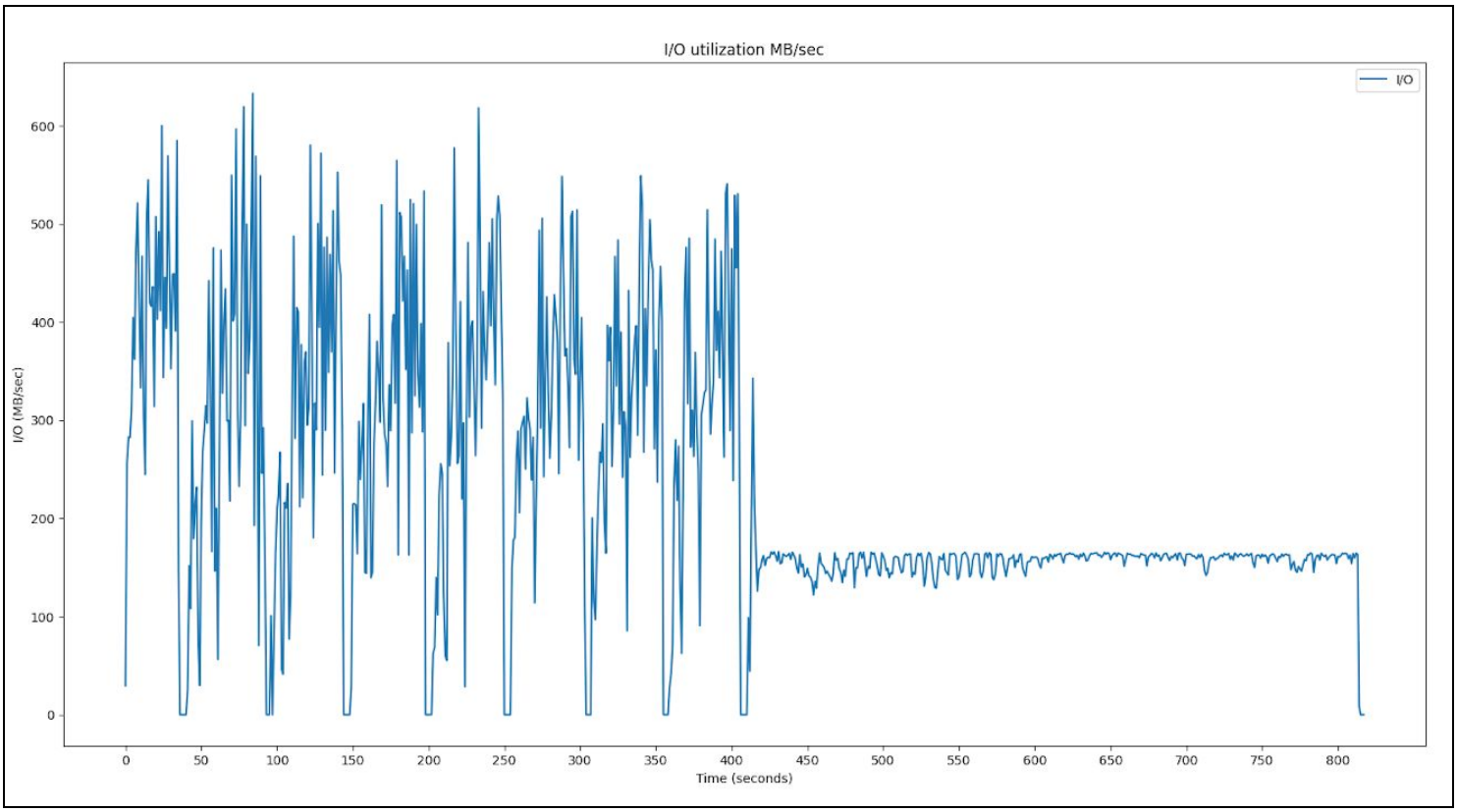

Fig 3. Overall I/O utilization Shared Memory

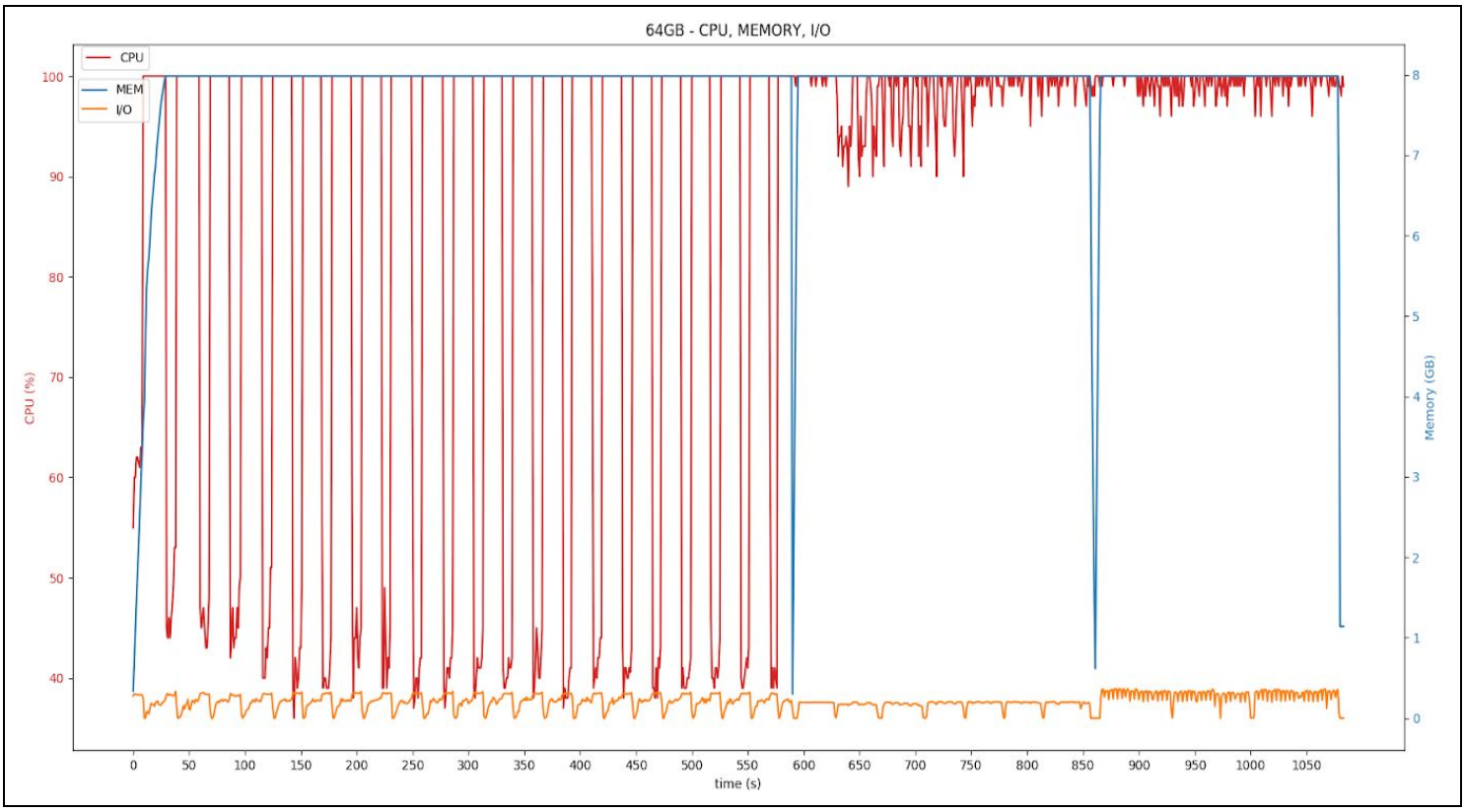

Figure 4: CPU vs I/O vs Memory utilization Linux Sort

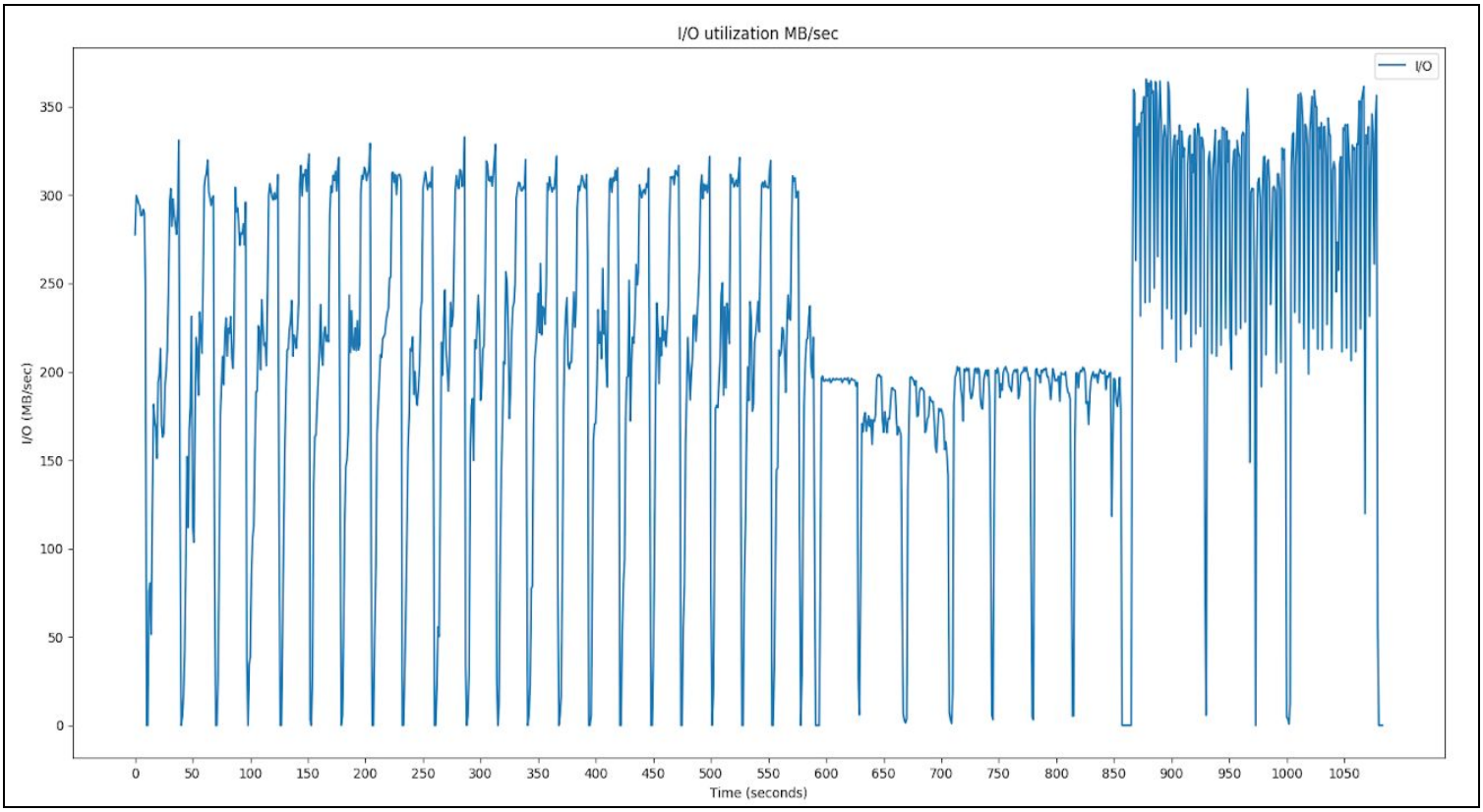

Figure 5: Overall I/O utilization Linux Sort