Project 2 Report

Suresh Babu Jothilingam
Nitish Krishna Ganesan

Introduction:

The goal of project 2 is to design and develop a distributed, scalable and fault tolerant algorithm to sort a large set of numbers which has a character prefix and compute the sum of the number which has the same character prefix (case insensitive).

Design of Communication Protocol:

Transmission Control Protocol (TCP) is used for the communication between the Master and Slaves. A thread will be running at each slave with an open port so that the Master can communicate and send the chunks to the Slaves. Multiple threads are created at the Master node to communicate with the corresponding slaves. These connections will remain active after the chunks are sent to the corresponding slaves. Once the slaves sort the chunks, the connections will be used by the slaves to send the sorted chunks back to the Master.

A Monitor thread will be continuously running at the Master node to check the progress of the Slave nodes. From time to time, the Monitor will ping the slave nodes to check the status. If the slave is dead, corresponding action will be taken by the Failure Handling module.

Load Balancing:

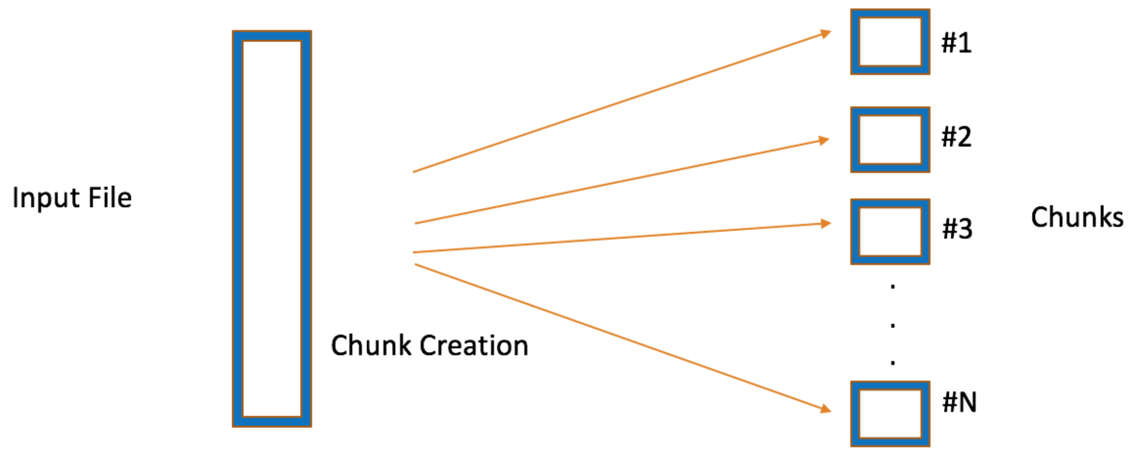
Load balancer is a system which is used to transfer the request from a Client to a Server by selecting the best available server. The best server is chosen by evaluating the server on different criteria like processing speed of the server, load of the server, requests waiting to be processed at the server end etc.

For our approach, we initially developed a load balancer which could get status of the slave node’s queue. The queue is where unprocessed chunks are placed at the slave node. The load balancer, based on the queue length chooses the best available slave. The chosen slave has smallest number of unprocessed chunks in the queue. Using the developed load balance there is an additional overhead at the Master node. It is because the load balancer has to get the queue status from all the active slaves, do analysis and choose best slave. We identified two issues with using a load balancer. The first issue is that in most of the cases not all the slaves were utilized. The second issue is, the load balances incurs some additional time to perform its task.

Based on the issues mentioned above we decided to restructure our design to work efficiently without using a load balancer. Before the chunk creation phase we decide the chunks to be sent to each slave based on the number of words in the input file. In the chunk distribution phase new thread is created for each slave and the created threads are responsible for distributing assigned chunks to the respective slaves. This approach works effective compared to the previous approach. Failure handling is also covered with this approach. Exception in a thread means the slave handled by that thread has failed. The dead slave is easily located and the chunks are retransmitted to an active slave.

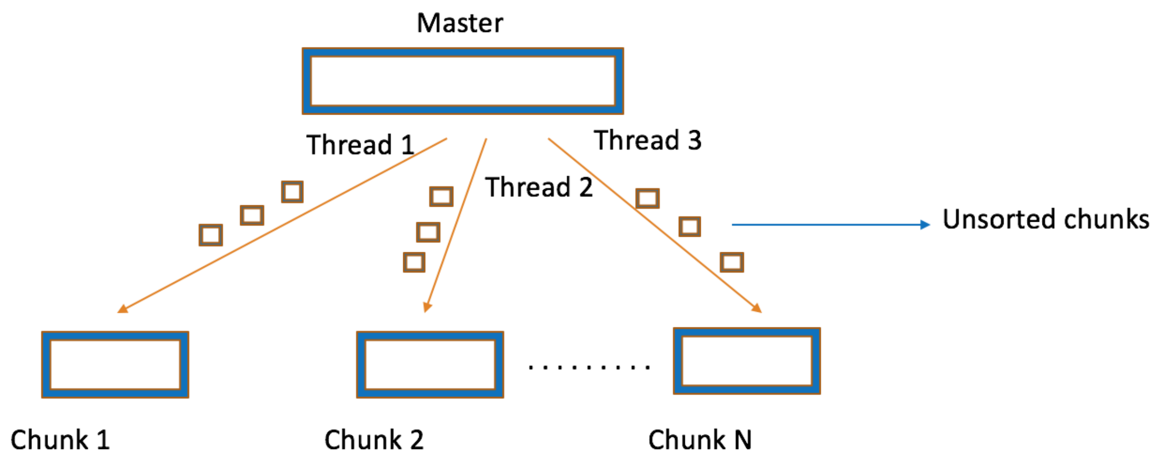
Implementation Ideas:

We focused on simple and reliable design. There are two initial phases in our approach: Chunk creation and Chunk distribution.



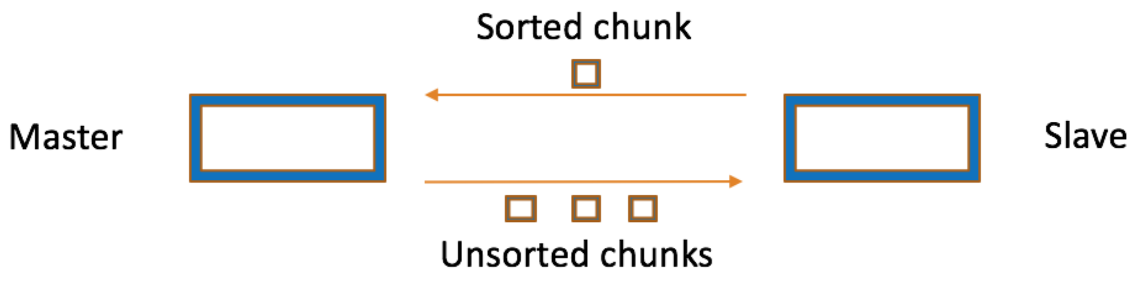
Chunk creation

In chunk creation phase the input file is divided into small chunks. In our algorithm each chunk will contain at most 100000 words in it. During the process unique ID is assigned to each chunk and it is mapped to a slave. We used Map data structure to store these details. The max size for a chunk was chosen after evaluating the performance of the system using different sized chunks. Once the chunks are created the input file is deleted in order to free up the memory and to make sure only one copy of the file is kept at the master node. We chose large chunk size because we noticed that merge takes lot of time when give lot of files.



Chunk distribution

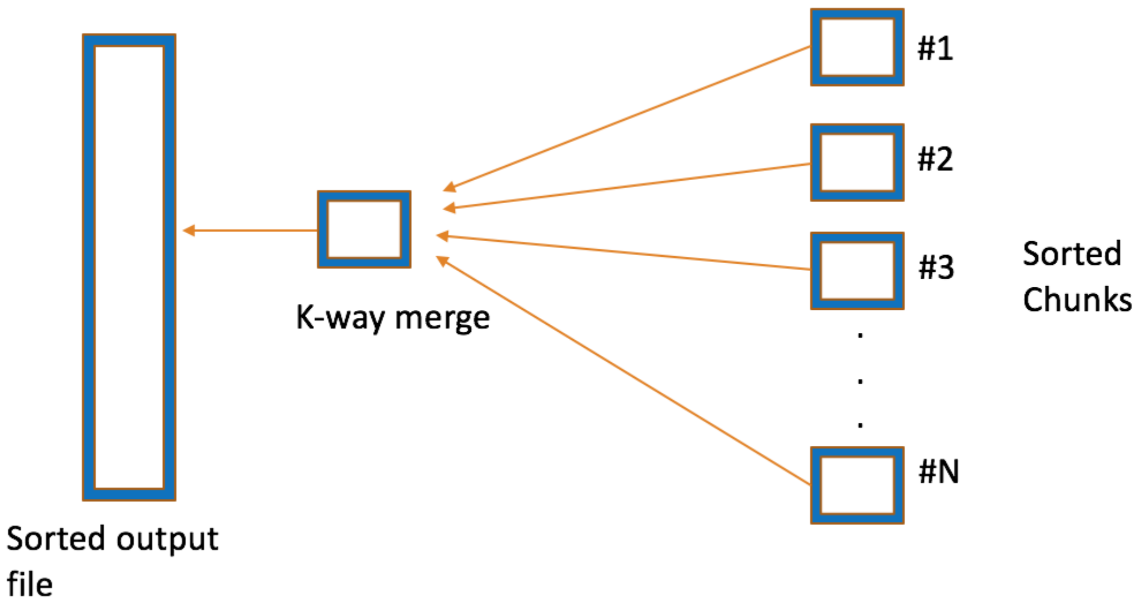
In chunk distribution phase, new thread is created for each slave. These threads are responsible to transfer chunks to respective slaves. During the chunk creation phase, chunks to be transferred to each slave is decided. File is read only once while creating chunks. When the required number of chunks are created for a slave new thread is created and the created chunks are transferred to the slave.



Receiving chunk from slave

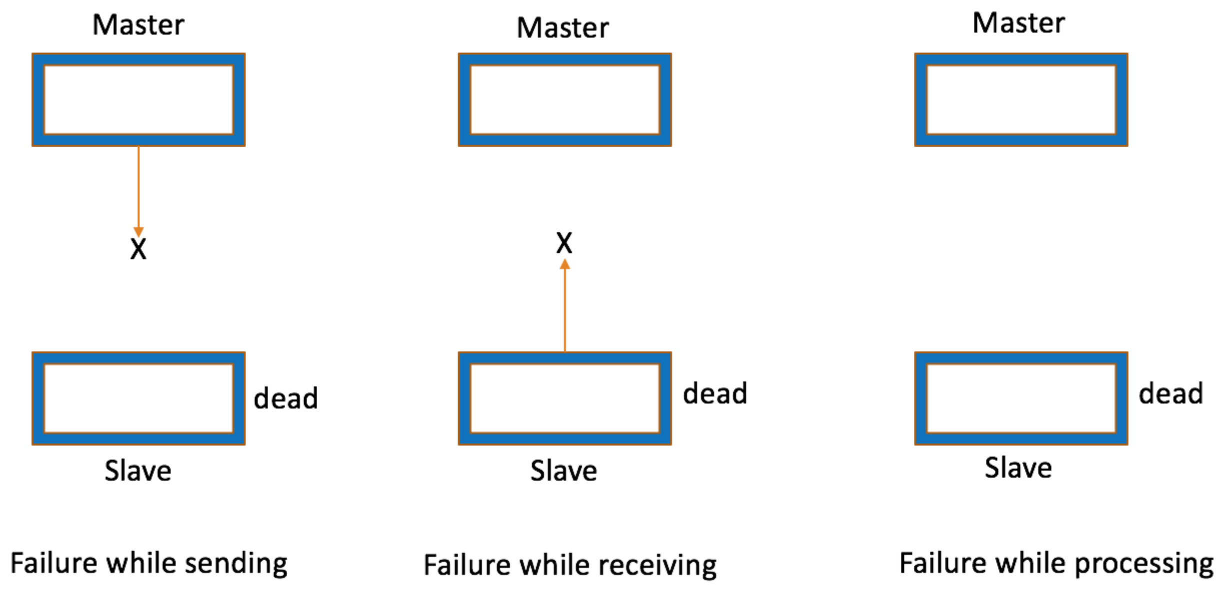
At the slave node, processing takes place after a complete chunk is received from Master. New thread is created for each received chunk. This approach does not block the processing of incoming chunks. As the chunks are processed it is sent back to the master node for the final step.

We have used modified quick sort algorithm to sort the received unsorted chunks at the slave node. Choosing a good pivot element is a challenge in quick sort algorithm. If the pivot element is not chosen properly, time taken to sort N elements would be O(N^2), whereas in normal case it would be O(N*log N). There are many solutions proposed to avoid worst case complexity of quick sort. We chose Median-Of-Three approach, which is proven to be effective. This approach works by choosing first, last and middle element, sort the chosen three elements and take the middle element as pivot. Doing so it is made sure that median element is always chosen as pivot element. Thus, avoiding the worst case complexity.



K-Way merge

After receiving all the sorted blocks at the master node a final K-way merge is performed on the sorted files. As the sorted files are received the corresponding unsorted file is deleted to free up memory. K-way merge algorithm is used to merge sorted K files to a single file. K file pointers are created for K files. The pointers are marked before reading a line. The values are compared and smallest value is written to a file. The file pointer which doesn't have the smallest value are set back to the marked position so that they can be compared again. Once a complete chunk is merged to the output file it will be removed to free up memory.



Failures

There are three possible failures: 1) Failure while sending chunk files to the slave 2) Failure while receiving sorted files from slave and 3) Failure while the blocks are processed in the slave before sending the data. We handle all these failures by using a generic synchronized method. The argument to this function is the IP address of the slave which has failed and the chunk IDs assigned to it. The slave IP is removed from the active slave list and the corresponding blocks are sent to another active slave adding the chunk IDs to the new chosen slave in the map. This method can handle more than one slave failure. Failure can happen while retransmitting and in that case the same method is called again. We also have a Monitor thread which keeps checking if the slave is active or not. It immediately calls the error handler once it finds a slave is dead. This approach makes the system to work effective against any type of failure. No active slave case is also handled in our algorithm. In this case, once master node detects that there are no slaves, the master node acts as a slave and starts processing chunks. In this case there will be only one slave node which is the master node itself.

Evaluation Results:

For evaluating our system performance we gave varying input file size and varying the number of slaves. The results are as below:

1) 10000 words (59kb)

No. of Slaves	**Time to Sort**	**Time to Merge**
5	1.5 sec	2 sec
3	3 sec	2 sec
2	4 sec	2 sec

2) 100000 words (5.9 mb)

No. of Slaves	**Time to Sort**	**Time to Merge**
5	25 sec	37 sec
3	1 min	1 min
2	1 min 15 sec	1 min

3) 1000000 words (58.9 mb)

No. of Slaves	**Time to Sort**	**Time to Merge**
5	19 min	28 min
3	24 min	34 min
2	28 min	33 min

We noticed that the time taken to sort is reduced when more number of slaves are added to the system.

Lessons Learnt:

We learnt a lot of new lessons while designing and developing this project. We also understood different problems which are tend to occur while developing distributed systems like fault tolerance, load balancing etc. This section discusses about some of the lessons learnt while developing this project.

There was a problem with JVM when we used Java inbuilt data structures to store and manipulate data. For example, initially we used array list to store data, divide and sent it across all the slaves. The process was tremendously fast but as the data size increased Java was not able to allocate space. As a result we saw Java memory heap exceeded error. To solve this issue we did all our processing using external files. We stored data in a file, divided the data into small chunk files, distributed across slaves and merged all the files. This approach is definitely slower than the previous but it eliminated the need for our system to depend on Java heap memory. We also had an issue with different versions of Java. We noticed that Java version is not same on the slave nodes. Java code compiled on higher version will not be executed in lower version. To address this issue we sent the whole Java code to slave nodes, compiled and executed in there. It overcomes the stated issue as the code is compiled using the available Java version. This makes our approach more robust.

Future Improvements:

The future improvement of our algorithm would be to implement Hadoop architecture for fault tolerance. In this approach, three copies of a block is sent to three random slave nodes. Three is default and we can change it any time. Since three copies are sent, even if one slave fails the other two slave nodes return the processed block. Redundancy is avoided by dropping the packets which is already received. In our approach we send one copy of chunk to each slave. In case of failure we send the chunks responsible for the dead slave to an active slave.

The next improvement would be to assign alphabets to each slave based on the number of slaves. For example, if there are two slaves, A – M prefixed words will be stored in slave 1 and N – Z prefixed words will be stored in slave 2. In this approach merging will be easy because we sequentially get data from slaves and append it. There is no dedicated merging task in this approach and hence it will be faster. In our approach we use K-way merge to merge individual sorted files which is time consuming.

The other improvement would be to develop an optimized merging algorithm. The reason for this improvement is, we have noticed that merging small chunk files take more time when compared to sorting.