Project- 2 Report: Distributed sort using Raspberry Pi

Ganesh Rajasekharan
Harsh Patil
Shikha Soni

TASK 1:

This project deals with sorting a very large file with a limited RAM capacity while using a master slave architecture. Distributed sorting of a very large file using Raspberry Pis was essentially a difficult task keeping the processing power of each pi in mind. We proposed two designs while implemented one of them.

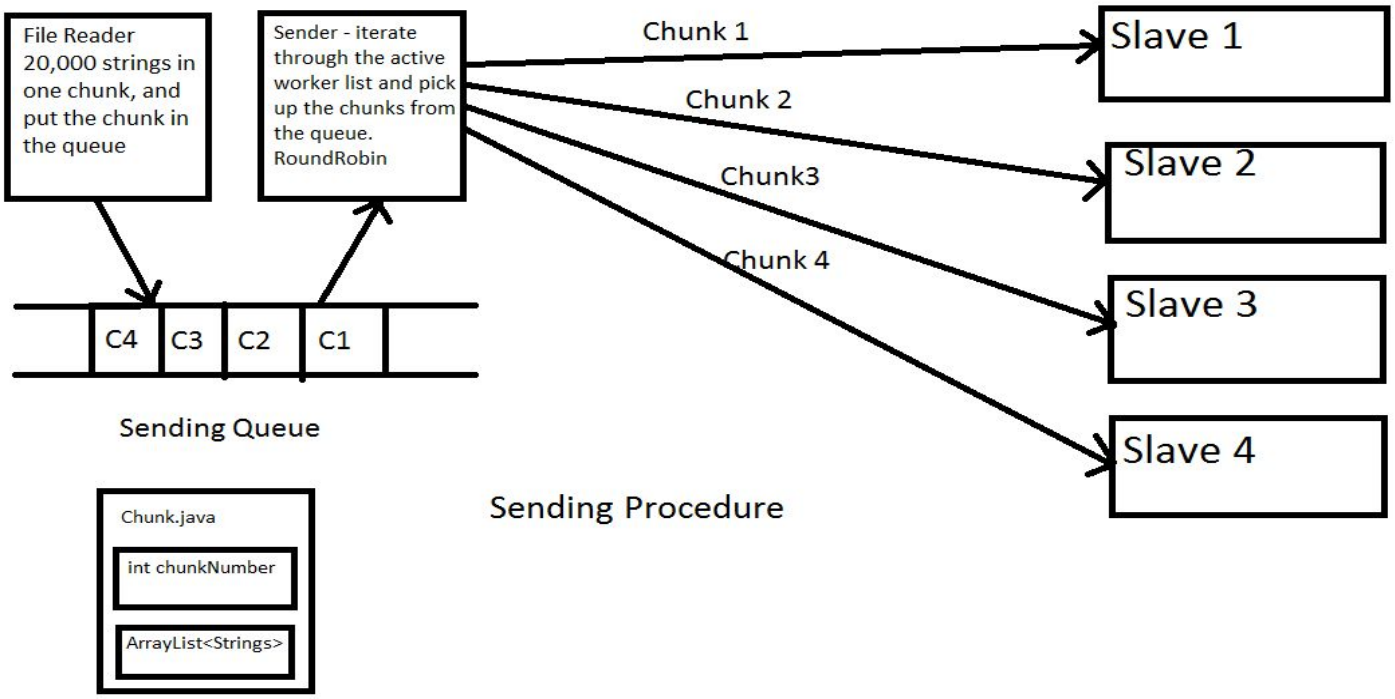
Design 1:

1. File Division:

Our implementation has a class Chunk. Every object of chunk class contains a chunk number and chunk content. The chunk number is the ID that each chunk has to be uniquely identified in the system, while chunk contents are the actual strings read from the big file. We decided to read 20,000 strings in one chunk from the file.

2. Chunk Assignment:

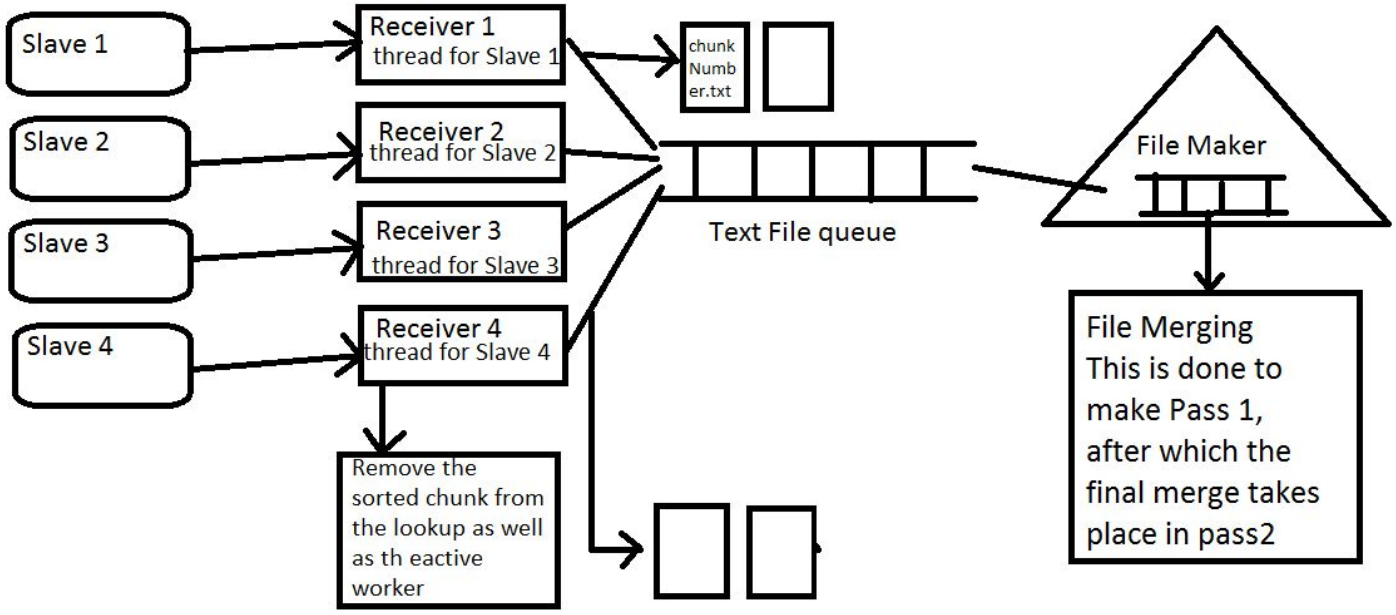
There are two threads that take care of reading and sending the strings. A reader thread reads the strings into a chunk and puts these chunks in a queue while the sender picks up these chunks and sends it to the available slaves in a round robin fashion.



3. Chunk receiving and merging:

The receiver on the master receives the sorted chunks and makes a text file out of it, while the file maker picks up 2*numPi (numPi is the number of available slaves) number of files, merges them into one and

makes a new text file which is Pass1 of external merge sort. The moment the merged file is created, the previous text files are deleted. The second round merges the next set of text files into one in a hierarchy. This merging of files is carried out until the received chunks Blocking Queue is empty and the total number of chunks sent matches the total chunks received. When this condition is satisfied the merge pass converges the files into a final sorted file.



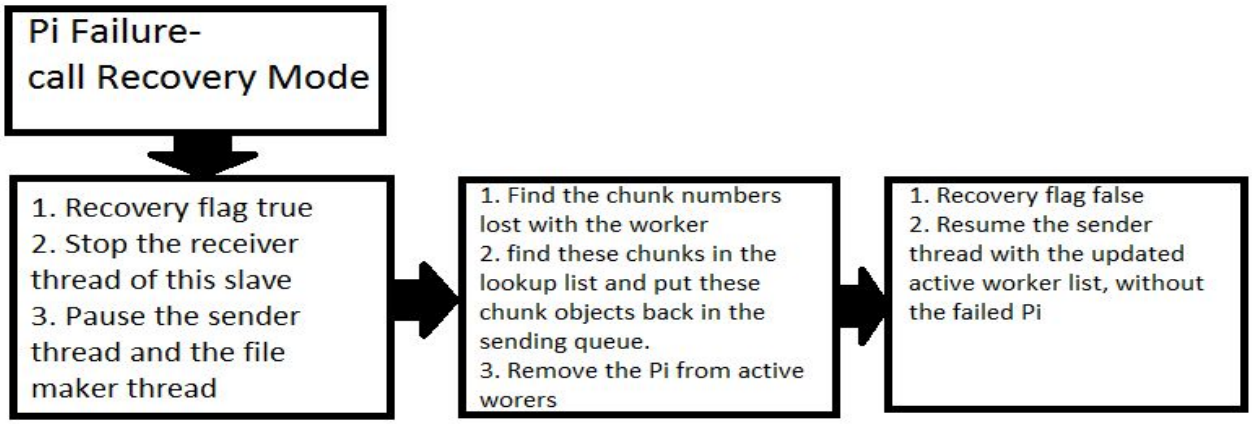
Receive Procedure

4. Fault Tolerance:

The moment the pi fails the system goes in a recovery mode. The system maintains a hash-map for active workers.

Key: Socket	Value: Chunk numbers	Look Up List
Pi1	1,4,8,16	Chunk 1
Pi2	2,5,9,17	Chunk2
Pi3	3,6,10,19	Chunk 3

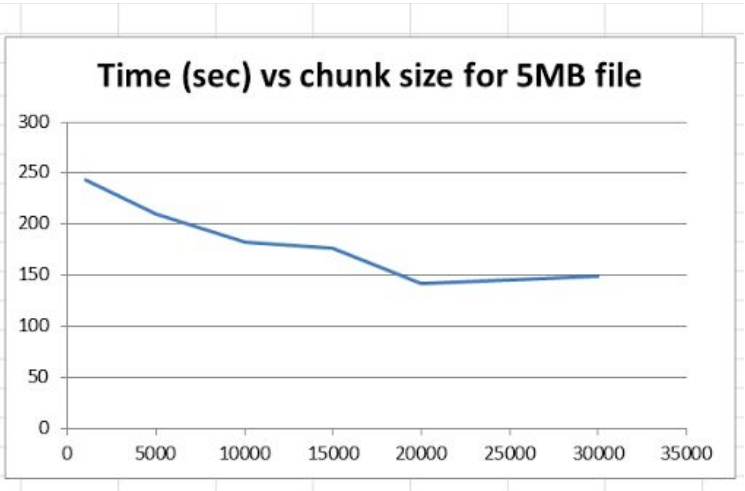
Also there is a lookup list that stores the chunk objects. The recovery mode verifies the socket that failed, finds the chunk numbers that are now missing checks the look up for these chunk numbers and puts them back in the sending queue.

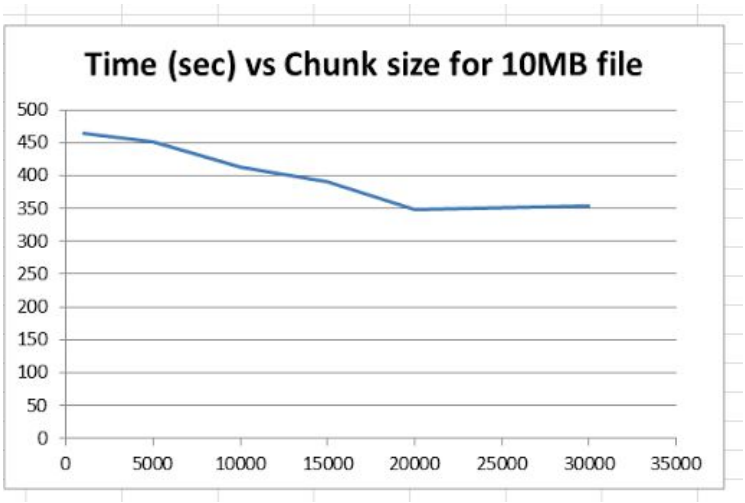


5. Observations

Design 1

File Size (MB)	Number of chunks	Time to complete (se
5	1000	700
5	5000	510
5	10000	505
5	15000	498
5	20000	515
5	30000	522
10	1000	1244
10	5000	1212
10	10000	1186
10	15000	1156
10	20000	1104
10	30000	1023
19	20000	46521





Design 2:

The file division and chunk assignment remains the same as the previous design, the merging step changes. Each of the slave doesn't return back the chunks sent. All the chunks sent are sorted and merged into one, and stored in the slave in a text file. Once all the chunks are done, all the Pis return back the big file. At the end of it, the master gets back 4 files that are merged into one by the master.

Fault tolerance:
Once the slave pi goes down, the script can make it run back again, and since the text file is stored in the memory of the slave it can send the sorted file back to the master. The master can then merge the files into one. The design implemented by us is the first one.
Important aspects to be considered :

1. **Selecting the File reader:** Since there is a constraint of time taken into consideration since there is high latency in the I/O operations of a raspberry Pi.

Consideration:
i. BufferredReader with varied buffer sizes.
ii. Memory Mapped Buffer
iii. Scanner
iv. File Channel
v. File Input Stream
We tried using all of these in order to calculate the time required to read the strings from a text file. The calculation showed that the file channel and the memory mapped buffer were the fastest to read. But since our input was a String for example A1234. While reading using bytes of data from the file, there is a high possibility that the string will be read half way. Hence we need checks that make sure that the string is read completely. The resulting time was pretty much the same as reading the file using a bufferred reader. Hence we decided to go with reader.next line and read each string line by line. We put 20,000 strings in each chunk. File Channel was also a good option but again reading the bytes needed additional checks.

2. **Selecting Networking protocol:**

i. TCP: TCP is slower in networking since it ensures that there is no packet loss, so the acknowledgements take time.
ii. UDP: We earlier decided to go with UDP since it faster and hence we can save on time but if there are packet losses it will be an additional overhead to take care of. Hence we decided to go with TCP.

3. **Data type to be stored in the chunk:**

i. Strings: Size: `8 * (int) ((((no chars) * 2) + 45) / 8)`
ii. Char[]: Size: 2* number of chars bytes
ArrayList of String objects: If there are 10 string objects: each string object, if has 8 characters will take 56 bytes, therefore 10*56= 56,000 + 16*10(header)+ 4*10(alignment)= 56200
Our initial decision was to read the chunks with an arraylist of string objects, but when we monitored the growing heap size, we noticed that since every string was stored as char array. In order to avoid this we came with the following solutions:
i. char[][]: a 2D char array.
ii. ArrayList<char[]>
iii. ArrayList<ArrayList<Character>>

After trying to use these approaches, the conditional checks took a lot more time and hence we were back to saving each string as a String object.

4. **Saving the heap size:**

i. Garbage Collector

ii. Clearing object references: LL AL

5. **Merging:**

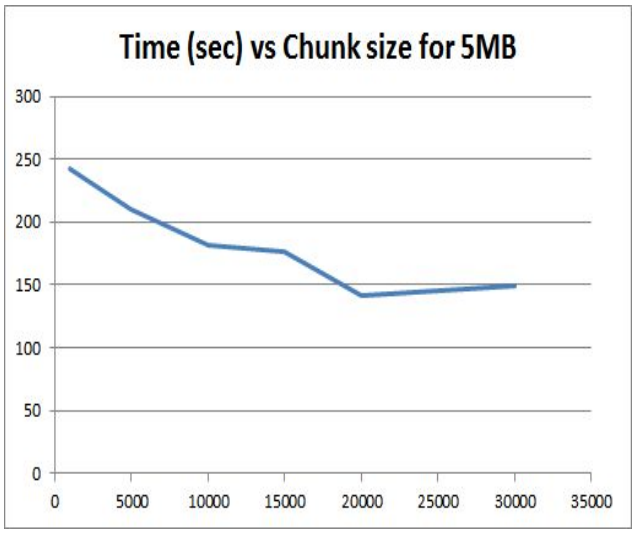
Using array of bufferedreader

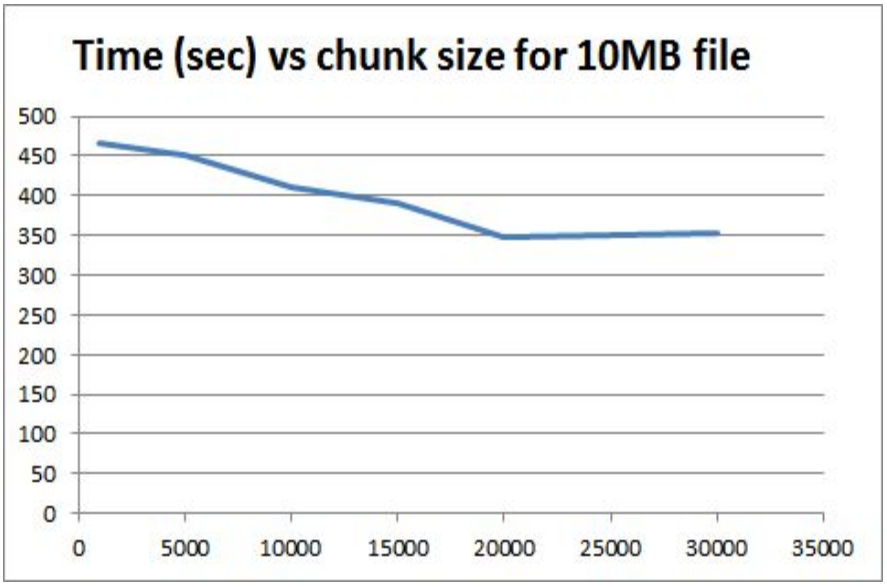
The top minimum values from the files to be merged are compared and the lowest value is inserted into the resulting merged file. This is similar to the merge step of the merge sort algorithm and takes O(n) for merging where n is the cumulative count of the strings to be merged. Since we maintaining bufferedreaders to each files to be merged and the final file the memory footprint is kept to a minimum which is affected by the buffer sizes maintained. This design is scalable for any amount of data in a limited RAM scenario since the disk storage is used to persist sorted files.

6. **Observations**

Design 2

File Size (MB)	Number of chunks	Time to complete (sec)
5	1000	243
5	5000	210
5	10000	182
5	15000	177
5	20000	142
5	30000	149
10	1000	465
10	5000	451
10	10000	412
10	15000	390
10	20000	349
10	30000	354
19	20000	27080





Challenges:

1. **Getting heap size under control:** After profiling the JVM heap size we were able to find out that the number of string objects were large and causing a degradation in performance. This was because string objects read from the input stream were not loosing references. By using a readUnshared method to read from the stream the table of references maintained by the ObjectInputStream were removed and resulted in saving heap size.
2. Fault tolerance challenges:
3. Minimizing the time on merge styles

TASK 2:

Requirement:

This part consisted of finding the frequency of occurrence of the character at the beginning of all the strings and the sum of the numerical part of the strings having the same prefix.

Implementation:

Our Master.sh script when started with a true flag carries out the task 2 by reading the input file line by line and adding the string to a HashMap<Character, Integer> which stores characters A-Z and a-z as keys and their frequency as the value. Also, another HashMap of type <Character, Long> is maintained where the first character prefix is stripped and the number is added as the value to the corresponding key with the prefix value. Once all the counts and sums are accumulated when the file has been read, a method is called to write the contents of the two HashMaps into a final file as required.