Distributed Computation on Raspberry Pi Network

Project Report

Authors: Amey Vikas Edwankar, Siddharth Rajendra Prasad Kalluru & Utkarsh Bhatia
Course: Topics in Systems
Instructor: Peizhao Hu
B. Thomas Golisano College of Computing & Information

***Abstract*— To create a communication protocol and load distribution algorithm that will enable us to sort a large dataset on a distributed network of Raspberry Pis. This exercise provides a learning environment which simulates the need of real world applications and problems in a classroom setting. This exercise would allow us to solve these issues in a distributed setting and simultaneously gain an insight in the field of distributed systems. The challenge also provides hands on experience in the facets of scalability, fault tolerance and transparency. Since the distributed systems contained of a network of Raspberry Pis there was an additional onus on creating an efficient algorithm that would be able to run in an environment that had constrained memory and power.**

I. Introduction

The amount of data shared online has increased manifold in recent times. To quote recent statistics, almost two hundred trillion gigabytes of data is shared online. Performing operations on data of this magnitude on one system is not possible, thus leading to the need of computation on multiple systems. This project aimed to make us understand and learn the importance of distributed computation in today's world. To simulate real world scenarios, a large dataset of N numbers needed to be sorted in a distributed environment made from Raspberry Pis. Simultaneously, an average of the unique numbers in the dataset also needed to be computed. The challenge was to complete the two given tasks in the least possible time. An additional issue to address was the constraint of memory and power due to the limited processing capabilities of the Raspberry Pi.

II. Methodologies

Due to the distributed nature of the system and low processing power of the nodes, normal sorting algorithms weren't viable candidates. Integrity of data across the entire system is a necessity, therefore normal partitioning, independent sorting and merging of the dataset would not lead to an accurate result.
To achieve the desired results a number of sorting algorithms were studied, their behavior in a distributed environment and their performance analysis with large data sets were compared. The algorithms that captured our attention were quicksort, comb sort, shell sort, insertion sort and a binary tree structure (basically used for searching algorithms).

To move ahead with the design and development of the project, it was decided to use a variation of quicksort algorithm which would be optimally suited to the given conditions.

A. Dual Pivot QuickSort

The first formulated approach was to use a dual pivot quicksort, this would enable the dataset to be partitioned into three sections bounded by the pivots. Two pivots in the dataset are chosen, the scope of these pivots is global and all the nodes have to partition elements in the dataset according to them.
The helper nodes are sent chunks of the dataset and the two pivots, the elements are partitioned into three files with respect to the pivots.
These files are sent to the master where they are aggregated, as shown in figure 1.

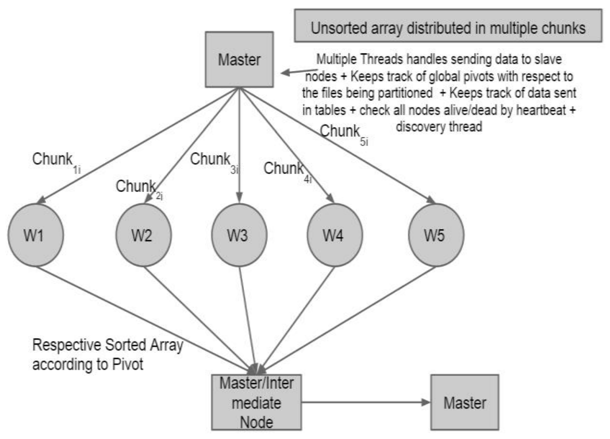
Unsorted array distributed in multiple chunks
Master
Multiple Threads handles sending data to slave nodes + Keeps track of global pivots with respect to the files being partitioned + Keeps track of data sent in tables + check all nodes alive/dead by heartbeat + discovery thread
Chunk 1i
Chunk 2i
Chunk 3i
Chunk 4i
Chunk 5i
W1
W2
W3
W4
W5
Respective Sorted Array according to Pivot
Master/Intermediate Node
Master

figure.1

Once the entire file has been partitioned, the partitions themselves are partitioned recursively until we reach the base case size decided for the file.
Once the base case is met, the file is sent to a helper where it is sorted using merge sort, shown in figure 2.

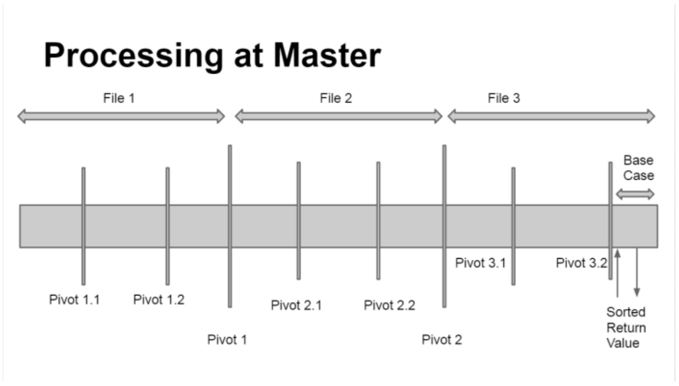

figure.2

After all the base case files are sorted they are merged together to create the sorted file.

The reason for choosing dual pivot quicksort was its time analysis compared to other quicksort algorithms as studied [1].

Random Data:

Quicksort Basic: 1222 ms

Quicksort 3 Way: 1295 ms

Quicksort Dual Pivot: 1066 ms

Duplicate Data:

Quicksort Basic: 378 ms

Quicksort 3 Way: 15 ms

Quicksort Dual Pivot: 6 ms

Findings: The above approach has some merits, since a dual pivot quicksort is being used, the dataset is being divided into three parts, since there is one more partition than the classic quicksort, this method facilitates the base case to be reached earlier and thus is faster at breaking down large datasets into smaller sets which can eventually be sorted much earlier than other methods allow. Therefore this method will prove to be highly efficient when used with large datasets with a large number of unique values.

However on implementation it was found that this method has very low performance on the hardware provided. The large number of file read and write operations makes this method inefficient on a Raspberry Pi due to the long time File I/O takes on an SD card.

Cons: Inefficient because of high file I/O.

Pros: May work for large data sets when master has high computational power.

B. Binary Insertion Sort

The first approach was evident in proving that File I/O must be avoided, hence there needed to be a change in the approach. Reading and writing from and to the SD card had to be kept to a minimum, therefore most of the processing had to be done in memory and the team had to devise a way to store the dataset in such a way that the dataset could be contained in the limited memory that a Raspberry Pi provides. The solution to this problem was achieved in the form of Java's Array List, since they are a dynamic array they allow all the benefits of arrays such as selection in constant time with the added benefit of it being growable. To make sure that the data would be contained within the Pi it was decided to use objects that contained key value pairs corresponding to the value of the element and the frequency of its occurrence. For this method the dataset would be partitioned into smaller chunks, added to an Array List and sent to a node along with a sorted Array List which contained the sorted objects according to the specification given above.

The helpers would traverse through the unsorted Array Lists elements and add them to the Sorted Array List after converting them into objects according to our specification.

If the current element is already present in the sorted Array List then only an increment of the frequency is required rather than creating a new object with the frequency set to 1.

Once the unsorted elements are added into the Array List, the helper returns the sorted Array List to the master, who keeps it in memory until another helper is available, in which case the master sends another unsorted Array List with the sorted Array List.

Once the entire file has been exhausted and all Helpers return the Array Lists to the master, the master merges them together and writes it to a file. The average is calculated by adding all the keys and dividing them by the size of the Array List.

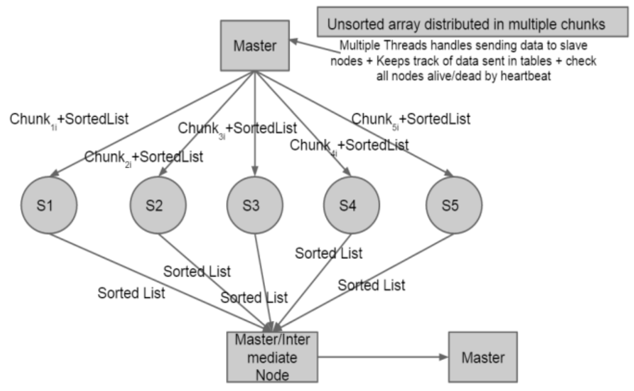

figure.3

Findings: This approach leads to a vast improvement in performance over the earlier method since reading and writing operations are only performed once and most of the operations are performed in memory.

The drawback of this approach is that a vast number of objects are created and they occupy a lot of memory, in addition to that there is an additional overhead since this method deals with user defined objects and therefore it'll be much slower than native collections such as a Tree Map containing integers as key value pairs.

C. Tree Map Design

Although the previous two approaches had very good asymptotic bounds, analyzing the time taken by them made it evident that they weren't the most optimal approaches for the restrictive processing power of the Raspberry Pis.

The third approach followed by the team used Java's Tree map, which is an integration of a binary tree and a hash map. As the input file was read by the master node for the first time after discovering a helper, the master makes a tree map unique for every helper profile, which was then sent to the helper as long as the helper node was active. This tree map was unique to every helper was sent to the respective helper along with the chunk of data read from the file, which is shown in figure 4.

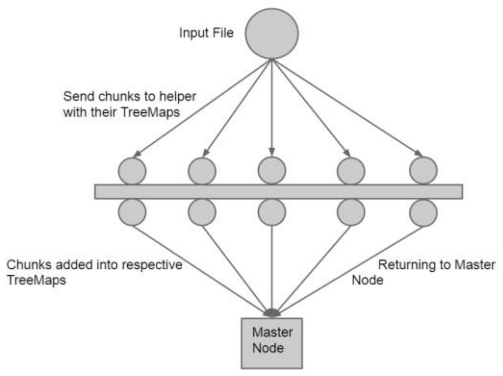
Input File
Send chunks to helper with their TreeMaps
Chunks added into respective TreeMaps
Returning to Master Node
Master Node

figure.4

The figure portrays that the respective helper nodes receive their tree map and the new chunk of elements. When the helper nodes have received the data, they add the new chunk of data into their tree map and send it back to the master node, The master retains a copy of the tree map to guarantee that no data loss can occur in the case of the helper failing, in this case the chunk is reassigned to another helper by integrating it with their own tree map and aggregating the data of both the tree maps.

When the file is exhausted and all the helpers have returned the sorted tree maps the master node affirms that no data has been lost because of network or node failure. After confirming, aggregation of data begins at the master node, by merging all the returned tree maps as shown in figure 5.

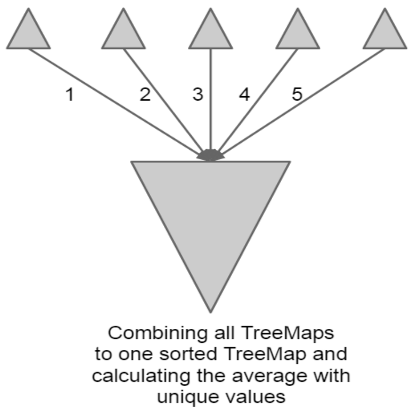
1
2
3
4
5
Combining all TreeMaps to one sorted TreeMap and calculating the average with unique values

figure.5

All the tree maps are merged in one, and this tree map is used to form a sorted file, the average is calculated with all unique numbers when the file is being written to an external file.

III. IMPLEMENTATION

After running performance tests on each of the three designs, the team decided to proceed with the Tree Map design approach. This particular approach performed best in fault tolerance, data handling & time performance.

A. *Master Pi*

The master Pi had the responsibility to assign work to all the helper Pis until all the chunks of the input file had been processed at least once at the helper, including all the dead chunks returned from the nodes which have crashed.

1) *Kickstart*

When the master Pi is initiated, the master requests the path for the input file and the directory where the user wants to place the sorted file. When the master has received the required inputs, it initiates the process, firstly it starts the assign chunk process which makes the first chunk from the input file and waits for a helper to join. At the same time another thread is started which keeps a track of the heartbeat of all the helper Pis which have joined. The master then awaits for communication from the helpers which are alive.

2) *Initialisation*

The script which starts all the helpers is run, the script starts all the helpers one by one. Individually, a helper's heartbeat is initiated first and then the helper process is started. This helps the master to lookup the helper heartbeat as soon as the helper notifies the master of its alive status. At the master node, a hash map is maintained which keeps track of all the helpers, as soon as a helper notifies the master, it adds it to the helper profile table map, which stores the helper's IP as its key and its helper profile as the respective helper profile.

3) *Helper Alive and Helper Profile*

When the master receives a notification from a helper the first time, the master proceeds to mark the helper as alive and creates a helper profile, which includes making a unique tree map for every helper, creating a list of objects assigned to that helper currently. A Boolean flag keeps track of the helpers return status for the current chunk assigned to it. The master also stores the two port numbers, assign port and heartbeat port respectively. The assign port is used to contact the respective helper to send the next chunk to the helper when it becomes available after processing the previous chunk to it, and the heartbeat port is used to constantly check the alive status for the helper.

4) *HeartBeat*

This is a separate thread which behaves like a daemon thread on the master Pi, it constantly runs and pings the helpers by referring to the Helper hash map which stores the helper details and the IP. If the helper goes down, it then proceeds to remove it from the helper profile map and checks other details regarding the respective helper.

5) *Assign Chunks*

When the assign chunk is initialized it creates a chunk of 100,000 objects read from the input file, this chunk of the objects is then assigned to an alive helper; we ascertain the status of a helper by referring to the helper profile table. Moving forward, the assign chunks proceeds to create subsequent chunks of 100,000 objects from the input file and waits for the next helper or the previous helper to become available, depending on the number of helpers which have joined the network and notified the master node about their presence. Simultaneously, the assign chunks process does a double check on the dead list before proceeding to create the next chunk of data from the file. If the dead list is not empty, the assign chunks process waits for the next helper to become available, merges the tree map of dead helper from the dead list and refers to the Boolean flag of the dead helper; if the helper had processed and appended the list to its own tree map then the assign chunks will proceed to create the next chunk from the input file. If the result had not been delivered back from the helper to the master, it then reassigns the sent list to the new helper. This assures that all the chunks are processed before the master starts proceeding to stage two.

6) Send Chunk

When a helper has been assigned a chunk, the assign chunk starts a new thread for every chunk assigned, so that it can carry out the tasks for the next chunk and all the helpers can work on their tree maps in parallel. The send chunk process sends the assigned chunk along with the helper tree map to the helper. The helper at its own end appends the sent chunk to its tree map and returns the modified tree map to the master Pi. When the master sends the chunk to the helper it sets its Boolean flag as false, and waits for the helper to return the modified list. If the helper returns the tree map after completing the process it then sets the Boolean flag as true so that the helper can become available to be assigned the next task in queue.

7) Dead list

When the assign chunk has read the complete file and assigned all the chunks the master Pi then proceeds to merge the returned input from the helpers. However, the master Pi does a check on the dead list before starting the merging process. It checks if the dead list queue is empty. If the dead list is not empty, it then proceeds to assign all the remaining elements of the dead list to the available helpers. The master does not start the merging until it has confirmed that all the chunks have been processed and appended in the tree map of a helper. This helps the master tackle fault tolerance in case a helper crashes due to unforeseen circumstances and assign the data of the dead helper to another helper which is alive and running.

8) Bookkeeping

To maintain a database of each alive helper in the given network, we have the helper ping the master node as soon as the helper becomes alive in the network. As soon as the master is notified that the helper is alive the master creates a helper profile and adds it to the helper profile hash map, which contains the key as the helper IP and the value as the respective helper profile details. The helper profile details consisted of a Boolean variable indicating whether the list that was sent to the respective helper has been returned or not, if false then which list (list amounts to the chunk of data extracted from the input file) has been sent to the helper. It also contains the information about the port number where the helper process is running and where the helper heartbeat is running. The helper heartbeat is constantly used to inform the master whether the specified helper is alive or not. If the helper dies, it updates the helper hash map and declares the helper as dead to the master process, which then checks the helper profile and sees if it was assigned a list which hasn't been returned, if the list wasn't returned by the helper it is then assigned to another helper which is alive and working. This is shown in figure 6.

9) Merging

When the master has confirmed that all helpers have returned their respective chunks and the dead list is empty, the master proceeds to merge all the returned tree maps into one tree map. The resultant master tree map contains information of the complete input file in a sorted form.

10) Sorted Output and Average

The master tree map is used to create a sorted output file at the directory specified by the user. While creating the sorted output file, the master also calculates the total of all the unique numbers, which is then used to calculate the average.

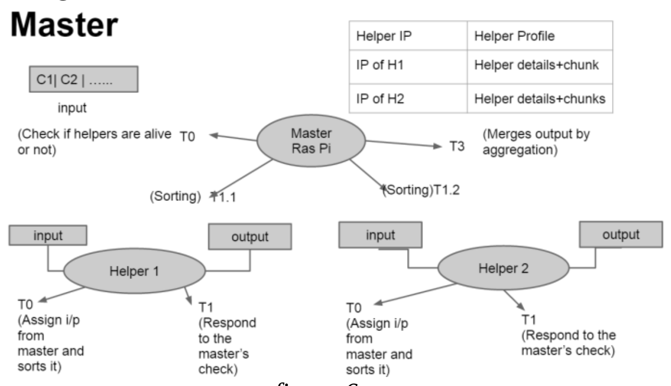
Master
Helper IP
Helper Profile
IP of H1
Helper details+chunk
IP of H2
Helper details+chunks
C1| C2 | ......
input
(Check if helpers are alive or not)
T0
Master Ras Pi
T3
(Merges output by aggregation)
(Sorting)
T1.1
(Sorting)T1.2
input
output
Helper 1
T0 (Assign i/p from master and sorts it)
T1 (Respond to the master's check)
input
output
Helper 2
T0 (Assign i/p from master and sorts it)
T1 (Respond to the master's check)

figure.6

B. *Helper Pis*

The helper Pis are used to merge the current sent chunks. When the assigned chunks are sent to the helper Pi from the master, the master also sends an intermediate tree map to the given helper Pi.

1) Appending

The helper traverses the sent list to it, and starts appending to the tree map received by it. When the helper has traversed the complete list and appended it to the given tree map it then sends the modified tree map back to the master node.

C. *Step by Step Algorithm*

1) The master process is initiated at the master node. When the master is started it asks for the path of the input file, it also asks the user to specify the path where he wants the master node to place the sorted file once the master has finished all its work.

2) As soon as the master has received the details from the user, it starts a heartbeat thread which listens to heartbeats from all the helper nodes it has registered as alive and *working. At the same time it will also start the assign chunks process which deals with assigning chunks to alive helpers.*

3) When any helper comes alive, the first task it does is to inform the master that it has entered the network and is ready to be assigned tasks, for this it calls the helper alive method at the master node.

4) When any helper initiates the helper alive method at the master node, the master enters the details which it receives from the helper and creates a helper profile respective to that helper. All the helper profiles are unique.

5) The helper profile contains all the information about a particular helper, such as its IP, its assignport, its heartbeat port, chunk assign status, chunk return status and the data of the chunk currently assigned to it.

6) Parallely, the heartbeat process, running at the master node, starts to ping the helpers which have been registered, to check if they are still alive.

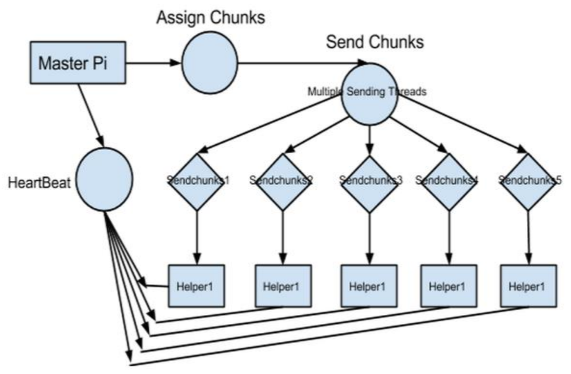

figure.7

7) When the profile has been created for a helper, the master node checks the deadlist queue, which maintains a record of chunks which have failed to return from the helper to which they were assigned to. This failure can happen in case the delay in the network was overwhelming or if that particular helper node has failed. In such a case, the details of that helper are stored in a queue which is extracted and aggregated with the helper details of another alive and available helper.

8) When the chunk is made by the assignchunk process, the master looks for a helper which is alive and available. The available status of a helper is indicated by looking at the helper profile details which denotes if the helper is already working on a chunk assigned to it or not. If the helper has processed and returned the chunk then the helper status is reset and it is now available to the master to be reassigned another chunk of data read from the file.

9) When the chunk is received at the helper node with the unique helper TreeMap for that helper, it processes the chunk and adds it to the TreeMap which it received from the master node and when the complete chunk of data is added into the TreeMap, it is then returned to the master node.

10) When the master confirms the return of the TreeMap from the helper, it marks the current chunk process for that thread as complete and proceeds to assign the now available helper the next available chunk as described in step 7.

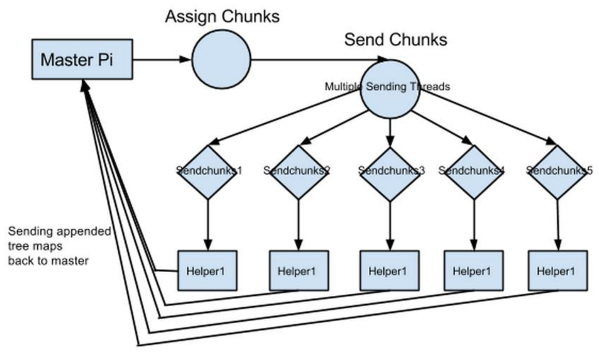

figure.8

11) This cycle is repeated until the master has read the complete input file and all the deadlist chunks with their profile details have also been assigned to alive helpers.

12) At this stage, the master reconfirms that all the available chunks of the input file have been processed at some stage in the assign chunk process and now there are no more remaining chunks to be assigned to any helper. When this has been confirmed the master node ends the assign chunks process and proceeds to aggregate all the data received.

13) The master node combines all the individual Tree maps from all the helper profiles and forms a master Tree Map which contains details of the complete sorted data.

14) The master then uses this TreeMap to form an external file in which data is written in sorted manner, and at the same time an addition is made of all unique numbers encountered while traversing the master TreeMap.

15) After the traversal of the master TreeMap is completed the sorted external file is saved, and the addition of all unique numbers is used to calculate the average of the input file which is then displayed to the user.

16) At this level, the master has completed all its processes, and the sorted file has been stored at the output path specified by the user and the user is also aware about the average of the input file and the master node ends its process.

IV. Performance Analysis

The performance analysis for the three tested approaches, Dual Pivot Quicksort, Binary Insertion sort and Tree map was studied and the following results were found.

Helpers	1	2	3	4	5
Dual Pivot	20	17.3	16.55	15.3	15
Binary Insertion	1.35	1.32	1.3	1.28	1.26
Tree Map	1.03	1	0.59	0.57	0.55

table.1: Performance as measured in number of helpers vs time in minutes

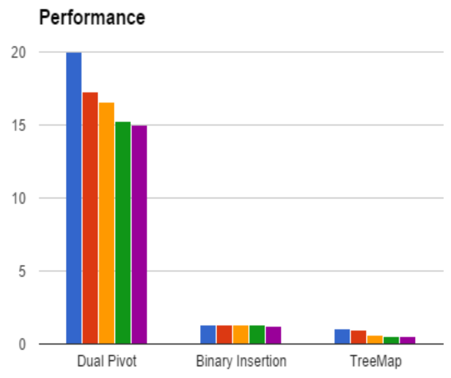

figure.10

Legend
Blue: 1 Worker
Orange: 2 Workers
Yellow: 3 Workers
Green: 4 Workers
Purple: 5 Workers

Also noticed during the test results was that if we sent a chunk of size less than 100,000 then the overall time consumed for the computation was large, and if a chunk of larger than 100,00 was sent then it increased the network overhead and file overhead in case of Dual Pivot Quicksort, and mildly affected the other two designs. Thus, the chunk size of 100,000 was chosen and gave the best performance analysis in the given environment settings.

V. Challenges

1) Scalability

The challenge of scalability was solved in all the given designs and in the implemented algorithm as well. There is no limit on the number of Helper Pis which can join the network. The master can also produce multiple chunks depending on the number of helper nodes available and registered alive to it. This would help the master node to compute the process on large data sets in less time, and would improve the overall performance of the system. In case of very large data sets ~1GB, when multiple tree maps are formed and are quite big, helper nodes can also be used to merge multiple tree maps until a small number of tree maps are left which can be merged at the master node itself and an output can be generated.

2) Hetrogeneity

The given distributed setting and the implementation design is based on the principle that it can be run on any system, irrespective of its operating system and hardware. This makes it easier to expand the network since it is not constrained by software and hardware.

3) Transparency

This is one of the main goals for a distributed environment. The end user is oblivious to all the back end processes and only experiences the end result, i.e. the sorted output file and the location specified by him and the average of all the unique numbers in the input file.

4) Fault Tolerance

The given implementation handles fault tolerance and makes sure that every read chunk from the input file is processed before the master node proceeds to the next stage and starts merging the tree maps. This was one of the main challenges of the given setting, and to test it a devil script was constantly running in the background knocking different nodes randomly. The implementation was tested in the environment and it was analyzed that the master node kept a dead list of all the helper which were knocked off and then assigned them to the next available helper, and merged the tree map with the next available helper as well. When the helper became alive again it was assigned a new tree map and could start processing the next chunk with a fresh tree map and no data was lost.

5) Memory and Power

As observed during the implementation of the first design, i.e. Dual pivot quicksort, the given distributed environment had a constraint on the memory and power it could process at a given time. The first approach used multiple file I/O and was faster when the master resided on a traditional device such as a laptop or a personal computer. Due to the Raspberry Pi's, memory and processing power limitations, the system performed sub optimally in the given environment and the team had to shift to different approaches which did not use extensive file operations.

VI. Conclusion

Although sorting is a general and widespread operation it needs a specific approach in this case where there are restraints on memory and processing power. The first approach where a dual pivot quicksort was used to partition data has multiple benefits, especially with large datasets but

for the constrained environment that the Raspberry Pi provides this method is lacking since it relies on a lot of File I/O.
It is necessary to minimize File I/O for higher performance therefore it is beneficial to use as much on board memory as possible.
Binary insertion and Tree Map allow significant increases in performance, with Tree Map allowing the best performance since it uses Java's inbuilt Tree Map and uses primitive instead of user defined objects which are not as efficient.

VII. Future Work

The current system works on the assumption that the file will not contain more than 100,000 unique elements. The performance and scope of the system can be increased in myriad ways. For example partitioning the file into larger chunks before sending them over the network will help us process the data quicker and ease the traffic over the network.
The current implementation uses RMI which is not as efficient as using sockets. Using sockets in a future iteration may lead to better performance as they provide more control due to their lower level nature.
For larger datasets ~5 GB the dual pivot quicksort will lead to better performance considering the environment contains higher processing power. The system can in future also a include a mobile discovery protocol which may discover idle devices such as mobiles and tablets connected to the same local network and allocate them tasks so that it can use their resources when they are idle.

VIII. Result

In the process of completing this challenge, the team learnt a lot about how a distributed approach can allow for deconstructing a large task into smaller tasks which can be easily solved by multiple helpers. Requiring to change the approach multiple times exposed the team to different problems and how to solve them. Encountering all the challenges, the team was able to demonstrate the implemented design and perform the required computation as initially required by the challenge.

IX. References

1) *http://rerun.me/2013/06/13/quicksorting-3-way-and-dual-pivot/*
2) *http://www.academia.edu/1976253/Selection_of_Best_Sorting_Algorithm*
3) *http://buffered.io/posts/sorting-algorithms-the-comb-sort/*
4) *http://jeffreystedfast.blogspot.com/2007/02/binary-insertion-sort.html*
5) *http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf*