Sorting N Numbers in Distributed Computation Environment of Raspberry Pi

Karan Shah Madhav Menon Sonam Padwal

May 2015

Abstract

In recent years, even though we have been able to achieve profound efficiency and speed to perform various tasks with the help of computers, everyday challenges are increasing exponentially making it indispensable for us to use this computing power collaboratively. In this project, we have developed a distributed computation framework that could sort N large numbers and find the average o the same. This framework consists of a Master and many Slaves, which when put on a network could communicate with each other to accomplish a given huge task with Master as the controller and Slaves performing designated tasks. The processor used in this project is Raspberry Pi

Introduction

In the recent years, computers have become more powerful increasing the storage capacity and processor power day by day. But there is a limit to this increase. As the internet population grows, the internet users are increasing geographically and companies like Amazon, Ebay, etc need server that would allow better and faster response time. As the load increases, the server needs a mechanism allowing parallel and faster processing. The concept of Hadoop i.e. Master-Slave has been around for couple of years. Based on this, we are building a system that would allow processing the job on different slaves thereby saving time and decreasing response time indirectly improving turn around time. The prototype built is scalable to process a large input file containing integer numbers and also to incorporate multiple Slave nodes. The Master efficiently selects and sends the chunk of files to different slaves for processing thereby distributing the work load on the slaves. The processing time decreases allowing the master to process the files in significantly less time. The design handles slave failures and could be used as a proof of concept for implementation of real life application with improvements.

Architecture

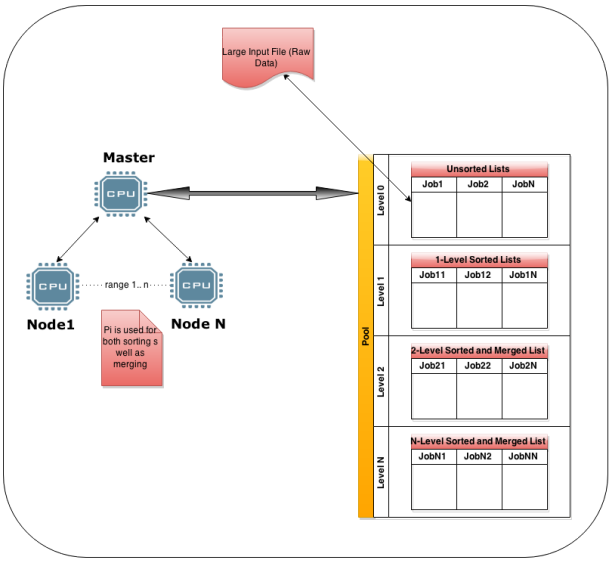

Figure 1: System Architecture

The distributed framework designed consists of Raspberry Pi as the computer node that communicates with each other and executes the tasks in parallel. One of these nodes act as a Master and rest of them act as Slaves. The Master is given the actual job to be performed which it divides into appropriate small tasks and puts it into a Job Queue. These tasks are then assigned to helper nodes i.e. slaves connected through Ethernet. The Slave is just a dumb node that performs the assigned task and does not keep any trace of the previous job. Once the assigned task is completed it sends back the result to the Master node and waits for the next job to be assigned by the Master. Thus, the Slave does not interact with the Job Queue as seen in the architecture diagram. Master interacts with the Job Queue through a JobManager. JobMnager maintains three different queues at each level namely joblist, interimlist and resultlist. Joblist is the one that contains jobs that are to be processed at that particular level. Resultlist is the list of the result files obtained from processing jobs. The Job Queue consists of several jobs to be executed at each level and is rest after every level to provide new jobs to the slave. The result files of the previous level are used as jobs at the next level. Finally, after reaching a definite size the transfer of files over the network is time consuming and unreliable and hence further jobs are processed at Master itself resulting in the final sorted file.

Job names at each level appends the job names it worked on from the previous level prefixed by the level it currently is. When files are split at the 0th level the jobs have numbers, 000 to 00x based on the number of files generated. So the files generated at the 1st level will have filenames like job1001 if it was

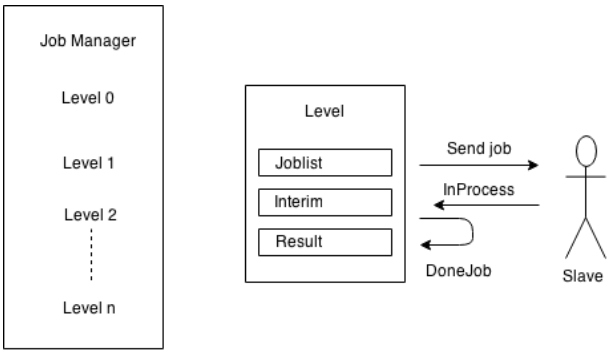

Figure 2: Job Queue Description

generated by sorting job001 and the subsequent levels will have jobnx. The jobs performed at the master will have prefix mas.

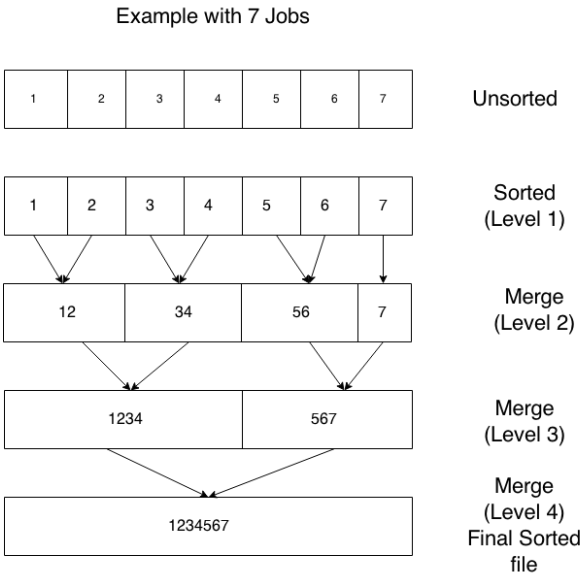

Figure 3: Job Levels Description

Implementation

The implementation is done in J2SE. The communication between Master and Slave is done using Java Secure Channel (JSch), which is a pure Java implementation of SSH2. The Master slave communication happens over RMI calls

which under the hood is actually maintaining sockets. Advantages of using RMI are as follows

- Simple and easy to implement and maintain
- Robust and flexible implementation
- Zero install for client users expediting the overall process time.
- Distributed system creations are allowed while disengaging the client and server objects at the same time.

The concurrent linked queue implementation is used to maintain and distribute jobs to the Slave nodes. It is an unbounded thread-safe queue based on linked nodes. This implementation uses an efficient "wait-free" algorithm when multiple threads try to access it simultaneously.

Sorting Mechanism

The Master is invoked and provided with the large job that is to be performed. In this case, a file containing integers on every new line is passed as an argument to the Master during its invocation. Next, the Master forks a thread Monitor Thread to connect all the available Slaves over the network and keep monitoring their connection after every specific interval.

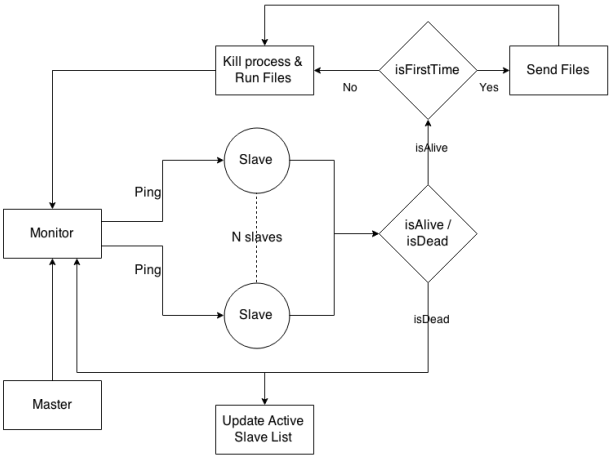

Figure 4: Monitor Thread Processing

As soon as the link between Master and every helper node is successfully established, the Master sends all the configuration files to the helper nodes using a secure file transfer protocol. The Master then runs these configuration files on Slave to starts the Slave processes, which pings back to the Master stating that it is ready to process a job. The Master main thread in the meantime divides the large job file into small files that are distributed to the connected Slaves. Now, Master contacts the JobManager to start sending files that he could assign to the Slave nodes.

In the first level, Master sends only one file that needs to be sorted. Every Slave receives the file sent by master, writes the contents into a file on its disk, sorts the assigned file, sends back the result to the master and deletes the temporary created file on disk to free the space. Master sends this result file to the JobManager, which manages all job queues. At Master, the Job Manager places resulting sorted file in the next level queue.

The next level is not started until all the files are sorted at previous level. Now, since we have every file that is sorted, the next level onwards we only have to merge the two input files resulting into one file. This process continues till we merge all the small files into one final result file.

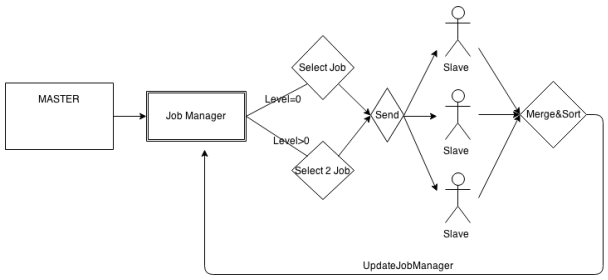

Figure 5: Job Processing Implementation

After certain levels, the files that are to be sent to the Slave for merging become large and hence their transmission over the network becomes time consuming and unreliable. Thus, after reaching this threshold, the Master takes over and stops the communication with the Slaves. Master then merges the rest of the files to form the final result.

Average Mechanism

To calculate average of unique numbers we need to find list of unique integers from this large dataset. In the first level, when the Slave receives the file of random integers the numbers are added to a Hashset at Slave end. Since the size of file is small enough to find out unique numbers this operation takes up very less time to find list of unique numbers. This list of numbers is also in parallel sent back to master to fill in a Hashset at Mater end. Thus, finding unique numbers and storing them at Master end is not done as a separate task but at the same time as sorting at the first level. At the end, the calculation of average is called upon to display the result on screen.

Scalability

The system we have designed works both for large as well as small data files. Also, the system would incorporate more Slave nodes to improve the efficiency. The system is being tested for a file as small as 350 bytes to 20 MB and also,

with one Slave node and five Slave nodes for processing. The statistics of how the system could scale both these parameters is evident from the graphs in the result section.

Fault Tolerance

The system reacts very effectively to the failure of nodes within the network. The monitoring thread which is constantly keeping track of the connection with the nodes intimates if any one of the connection is broken or dis-functional. The Master has a track of which jobs were assigned to Slave nodes and in such failure situation reallocates the job to another Slave. Here, as mentioned earlier, the interimlist is accessed to fetch the jobs assigned to failed nodes. In level one which is the sorting phase, the failed job is taken up by Master and completed. But in rest of the levels it is just that merging of the two files is to be done hence, when going to the next level, all jobs from the interimlist are inserted in resultlist. After switching to the next level, the previous level's resultlist jobs are put in joblist of this current level keeping the flow intact.

Results

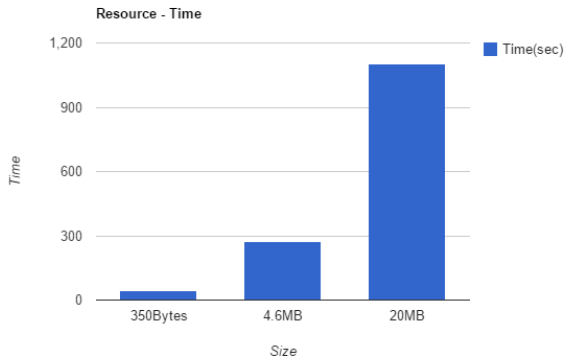

Figure 6: Scalability based on size of Input File

Lessons Learned

During the development phase, various functionalities had to be implemented in different way to design constraints. The RMI implementation had to be bi-directional since the file could not be sent back as a single obejct. Thus, MAster as well as server had to make a RMI call to each other for communication. This design was previously implemnted in different way where we ended up in a heap

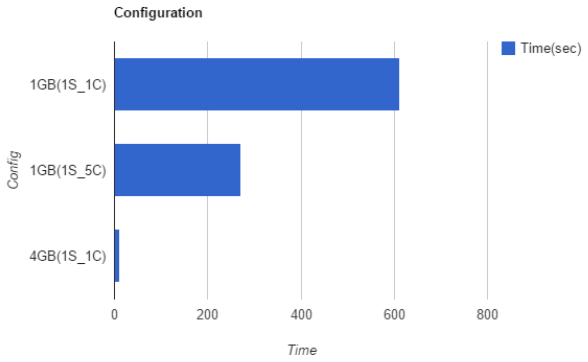

Figure 7: Scalability based on type and number of Processors as Slave

size exceeded error due to recurrence of calls to methods. This we later fixed by avoiding the recursive calls and increasing the modularity of the code.

Future Works

Current system could be incorporated with efficient methods to deal with the issues mostly due to time constraints. One of such factors is that when file is being sent to Slave node, the heap sort mechanism could be implemented for every input instead of writing entire data to one file and them sorting it. This could also save the input output operations improving the performance to a great extent. Also, initially Master sends and sets up all Slave nodes making them ready to process jobs. Instead invoking and configuring only required slave nodes would save up lot of time that is spent in configuration set up if the file is too small to be divided and sent to all slaves. Now, all Slave nodes interact with Master through Ethernet connectivity. This could be further extended to incorporate WiFi nodes too.

Conclusion

The system provides all the required functionality mentioned in the scope of the project. The system runs with a single script from start to end achieving the desired result. It has the capability of automatically detecting and starting the slave nodes and can handle node failure effectively. The choice of programming language along with the current design solves the problem of parallel and distributed execution of tasks as required.