A New Hadoop Scheduler Framework

Geetha J 1, N. Uday Bhaskar 2 and P. Chenna Reddy 3

Research Scholar1, Assistant Professor2 & 3Professor

1Department of Computer Science and Engineering, Jawaharlal Nehru Technological University Anantapur (JNTUA University), Anantapur, Andhra Pradesh, India.
2Department of Computer Science, Government College (Autonomous), Anantapur, Andhra Pradesh, India.
3Department of Computer Science and Engineering, JNTUA University, Anantapur, Andhra Pradesh, India.

© Research India Publications. http://www.ripublication.com

Abstract
Hadoop is an open source implementation of Google’s MapReduce framework which uses the Fair Scheduling algorithm. Fair scheduling is a method of assigning resources to applications such that all applications get, on average, an equal share of resources over time. This paper suggests a scheduler framework for Hadoop. The framework takes into account features like reduce task data localization, deadline constrained scheduling and localization of I/O intensive jobs and CPU intensive jobs. Many MapReduce applications come with strict deadlines and the fair scheduler doesn’t guarantee that the job will be completed within the given deadline. Given a system and its resources, our framework has the ability to decide whether a task can be completed within its specified deadline or not. It also makes sure that only the jobs for which deadlines can be met are forwarded to the scheduler, else there is a need to dynamically add nodes to complete the job within the specified deadline. Our framework also considers localization of resource complementary tasks to optimize resource usage which is otherwise ignored by Hadoop’s fair scheduler. For example, if an I/O intensive task is to be scheduled on to two slot node, a node with one slot occupied by a CPU intensive task is chosen and vice versa. Resource usage is maximum in this case. Finally, our framework also decreases local network traffic by scheduling reduce tasks in such a way that there is minimum data movement across the local network. In this way we make maximum use of the bandwidth available and enable quicker processing of data.

Keywords: Data Localization, MapReduce, JobTracker, TaskTracker

INTRODUCTION

In recent times we have seen a lot of advancement in the way people are connected and the way information is gathered and shared. Data is growing at a blistering rate. Major sources of data have been social media platforms, sensors data, machine log data and more. The data being unstructured adds to its complexity.

Hadoop is capable of processing such enormous and complex data. Hadoop is an open source framework used for distributed storage and processing very large structured and unstructured data sets over scalable set of computer nodes, each offering local computation and storage. A typical Hadoop cluster is a set of computers via LAN connection which follow Master-Slave architecture. The master node is called JobTracker and the slave nodes are called TaskTrackers [1]. In many big companies who use Hadoop extensively, the number of nodes in a cluster is usually in thousands. These Hadoop clusters are located in different data centres that are separated geographically. The processing unit of Hadoop is MapReduce [2]. The JobTracker splits the job into several maps and reduce tasks. These tasks run in parallel on the TaskTrackers. Map process converts the input to key-value (k,v) pairs. Shuffler process will sort the (k,v) pairs and this intermediate output is input for the reducer. Reducer will then combine the sorted pairs in way that it will solve the problem. A scheduling strategy is used to determine when a job can execute its tasks as Hadoop jobs have to share the cluster resources. Task scheduling in Hadoop is based on a pull-strategy. TaskTracker pulls the job by making requests to the JobTracker.

This paper proposes a scheduling framework. The framework optimizes three parameters - network traffic, job waiting time for the resources and maximizing the resource usage factor. It calculates the number of slots required for a task with a given deadline and hence it reduces the job’s wait for the resources. This framework enforces maximum resource usage and optimizes the resource division by which we can save significant amount of energy. By ensuring that reduce tasks run on a suitable TaskTracker such that it needs minimum bytes of data transfer across the network, congestion caused over network can be avoided.

METHODS

Deadline Scheduler

Consider a job, jobt that is currently being processed by Hadoop. Then jobt+1 is the job that is next in the job queue and has a deadline D. The framework runs the jobt+1 through
a schedulability test which determines the minimum number of resources \( N_{\text{min}} \). Job \( j+1 \) needs to be able to complete within deadline \( D \). If resources are available in the given system are greater than or equal to \( N_{\text{min}} \), job \( j+1 \) has passed the schedulability test and is pushed into the job queue. If the resources available are less than \( N_{\text{min}} \), the job \( j+1 \) has failed the schedulability test and the system, if possible, would scale accordingly to accommodate job \( j+1 \) or else would decline it[2].

Localisation of complementary tasks – Resource optimizer

If the job has passed the schedulability test, it has entered the job queue. In Fair scheduler of Hadoop, CPU and IO jobs are given equal share without considering their respective resource usages. Our framework slightly relaxes this fairness to achieve high resource usage. When the job is picked for processing, we determine the nature of the job, i.e., divide it into IO intensive tasks and CPU intensive tasks. A CPU intensive task would be assigned to a free slot on a slave for which the other slot would be occupied by an I/O intensive task. Vice versa for I/O intensive tasks [3]. These way complementary tasks are kept together on slave machines to maximize resource usage [4].

Minimal network congestion by data locality feature

Hadoop has a tree style network topology wherein nodes are distributed over different racks contained in many data centers. The network bandwidth is an important area of optimization. Nodes on the same rack have higher network bandwidth than nodes placed on different racks; similarly nodes in the same data center would have higher network bandwidth than nodes placed in different data centers. Once map tasks are scheduled, the nodes request the JobTracker to run reducer tasks. Our framework instructs the reduce task scheduler to perform reduce tasks on the node which has maximum size of intermediate output. This would minimize the intermediate output data transfer that happens between nodes and in turn reduces congestion in the network [5].

Figure 1. Explains the flow of a MapReduce job in the proposed scheduler framework. Any task which is submitted to the framework will first go through a test where the minimum number of required slots for the task is computed. Based on the availability of the slots the task is scheduled. Once the task is scheduled the resource optimizer assigns the tasks to slave nodes in a complementary way (CPU intensive and IO intensive task) to the slave nodes in order to achieve maximum resource usage. At first, the tasks are classified under CPU and IO intensive category. Next, the resource optimizer will identify the type of task running on the slave node which will help in assigning the complementary task. Resource optimizer is an improvement over the fair scheduler. Further, reduce task scheduling is enhanced with data locality feature where the reduce instance is assigned to the node where the maximum sized intermediate output is produced.

**DESIGN AND IMPLEMENTATION**

**Deadline scheduler**

We define a few terms for the derivation of the expression for min number of map and reduce slots required.

\[ q \rightarrow (D \ A \ \sigma) \] where \( q \) is a query consisting of \( D \) deadline. \( A \) is the arrival time and \( \sigma \) is the size of the input data.

\[ n \rightarrow \text{total slots assigned to a job, it is a total of n-map and n-reduce, ie,} \ n_m + n_r. \]

\( f \rightarrow \) fraction of input that a map process generates as output also called filter ratio.

\( \text{fr} \rightarrow \) The input to reducer or map output can be obtained by the product of filter ratio \( f \) and input size \( \sigma \).

\( J \rightarrow \) A Hadoop job which is a collection map and reduce tasks.

\( S_m \rightarrow \) Start time of the first map task for the job.

\( C_m \rightarrow \) Cost of processing a unit data in map task.

\( C_r \rightarrow \) Cost of processing a unit data in reduce task.

\( C_d \rightarrow \) Communication cost of transferring unit data.

To estimate duration of job \( J \), we take into consideration time taken to complete map tasks, time taken to complete reduce tasks and time taken for data transfer during the reduce copy phase. We also know that job time must not exceed the Deadline \( D \), and hence

\[ S + \sigma C_m/n_m + f_r C_r/n_r + f_c C_d \leq A + D \ldots |1|. \]

If we consider \( S_r \)-max to be the maximum value of reduce start time, which is the time taken for map task completion.

\[ S_{r_{\text{max}}} = A + D - f_r C_r/n_r -f_c C_d \]

We also know that

\[ S_m + \sigma C_m/n_m < S_{r_{\text{max}}} \]

Which gives
n_{\text{min}} = \sigma C m / S_{\text{max}} - S \ldots[2].

From 1 and 2, we get

n_{\text{rmin}} = \frac{r C r}{(A + D - r C d - S r)}

Deadline scheduler uses these criteria to schedule Hadoop Jobs with available deadline.

Algorithm:

Input: \( c \) - cluster
\( p \) - job pool

Output: whether the job can be scheduled or not

freemap slots = \( c \).freeMapSlots
freereduceslots = \( c \).freeReduceSlots

Repeat

\( j = \text{nextJob}(p) \)
arrivaltime = getStartTime(\( j \))
mapcostperunit = getMapCostPerUnit(\( j \))
reducecostperunit = getReduceCostPerUnit(\( j \))
shufflecostperunit = getShuffleCostPerUnit(\( j \))
reducestarttim = getReduceStartTime(\( j \))
inputsize = 0

Repeat

inputsize = inputsize + length(inputsplits)
Until (endOf(inputsplits))

Evaluate MaxReduceStartTime

if MaxReduceStartTime < arrivaltime
throw exception

Compute minMapSlots
Compute minReduceSlots

if minMapSlots > freemap slots or minReduceSlots > freereduceslots
return ConstraintFail
else
return ConstraintPass

Until (EndOf(jobs in Pool))

Resource optimizer

The current implementation of the Hadoop scheduler doesn’t take into account, the tasks and slave’s resource usage while scheduling jobs. CPU and IO jobs are given equal preference without considering their resource usage.

Figure 2: Comparison between Fair Scheduler and Fair Scheduler + Resource Optimizer

Figure 2. Shows allocation of slots for IO-intensive and CPU-intensive jobs on a slave by Fair Scheduler and Hadoop Scheduler Framework [6]. There are two parts to improve resource usage. One is to measure the tasks resource usage and the other is the slave’s resource usage.

Evaluate Task Resource Usage

We evaluate I/O and CPU usage for each running job before scheduling them. A MapReduce job normally is made of a few map tasks and a few reduces tasks. As each map task of a job would process nearly similar chunks of data, the map tasks of a job need roughly equal amount/number of resources.

Evaluation of the resource usage of Jobs

TaskTracker.java

We write a function “findCpuUtilization()” that executes a shell command to determine cpuUsage of the process from the system, which is specific to each task launched on the taskTracker. At the start of each task, take the pid of the task that is being launched on to the taskTracker. Then call the function findCpuUtilization() at the start of each task with respect to pid of the task. Store the information on to the task class of the task that has got the cpuUsage information.

Task.java

The task class has the information of the task being executed. The taskStatus variable in this class is updated according to the changes in the task.

TaskStatus.java

A private repository to store the information of the cpu utilisation. Accessors are defined for the above mentioned class.
JobInProgress.java

We define a repository to store job wise cpuUtilisation. The cpuUtilisation of 10% of the tasks of the specific job are taken to find if the job is Cpu intensive or I/O intensive. Static count for the currently running job is maintained and is reset when the job is initialized.

Slaves resource usage evaluation

We obtain the resource usage of each slave, if it is prepared to take up a new task. The evaluation of the slaves’ resource usage helps us to decide what kind of job is running on the slave currently, and what kind of job (CPU/IO) needs to be scheduled alongside. When a slot gets freed, our scheduler framework assigns a task whose resource requirement is the most complementary to the slave’s resource usage.

Evaluation of the resource usage of the slaves

TaskTracker.java

During every transmitHeartBeat() function call the cpuUtilisation of the TaskTracker is evaluated and sent to the JobTracker. Whenever the AssignNextTask condition is set, the CpuUtilisation of the taskTracker is evaluated and is stored so as to evaluate the cpuUsage in between the two times. This information represents the slave resource usage in the system. Then it is sent as the TaskTrackerStatus on to the jobTracker along with the heartbeat.

JobTracker.java

The information is received in the jobTracker and is forwarded to the TaskTracker.

FairScheduler.java

Information received from the JobTracker is used to determine if the TaskTracker that currently has a free slot has a CPU intensive or I/O intensive job running. The FairScheduler has a list of Schedulables from which the next tasks are to be scheduled on to the free slots in the TaskTracker. This calls a FairShareComparator of the class SchedulingAlgorithms to select among two of the Schedulables.

SchedulingAlgorithms.java

The scheduling algorithm has a class called FairShareComparator, which implements the FairScheduling algorithm. The needy job (Schedulable) that is complementary to the current state of the slave that requested a task is chosen to be scheduled.

Data locality

Consider the scenario as shown in Figure 3. Node 1 holds 5 MB IO: R1 (intermediate output for reducer 1) and 10 MB IO: R2 (intermediate output for reducer 2). While Node 2 holds 2 MB IO: R1 and 26 MB IO: R2. Node 3 holds 15 MB IO: R1 and 1 MB IO: R2. Once the map tasks are scheduled, Node 2 requests the JobTracker for permission to run reducer 1. As a result 20 MB of intermediate output must be transferred over the network. On the other hand node 1 requests to run reducer 2. As a result 27 MB must be moved to node 1. Hence, a total of 47 MB utilizes the cluster bandwidth. If node 1 or node 2 is situated in different racks compared to other nodes more congestion in the network will be prevalent. If this scenario were modified, such that reducer 1 ran on node 3 and reducer 2 ran on node 2, then 7 MB and 11 MB of data would be shuffled. Clearly, this result in an improvement of 50% reduction in the number of bytes shuffled. Hadoop’s present reduce task scheduler is incapable of making such decisions.

Figure 3: Scheduling Reduce Tasks in native Hadoop

We have implemented our method on Hadoop 1.0.3 as follows:

Formulation of a data structure that keeps track of the size of intermediate output generated by a mapper for every reducer - There are two in-built features of Hadoop that are exploited to gather the input required by the reduce task scheduler: Index File and Heartbeat protocol. The size of the intermediate output generated by a map task is available in the current versions of Hadoop. However, the manner in which the intermediate output is intended to be divided among the reducers is necessary. This information is available in the index file and is stored in the local file system of every TaskTracker. Heartbeat is a mechanism for a TaskTracker to announce periodic availability to the JobTracker. Besides the heartbeat, the TaskTrackers send information regarding its
states to the JobTracker. By making modifications in appropriate classes, the JobTracker collects the index file information from the local file system of the TaskTracker via the information sent along with the heartbeat message.

Controlling the scheduling of reduce tasks - Using the generated data structure, the TaskTracker that holds the largest intermediate output for a particular reduce task is determined. Upon which the reduce task scheduler takes the requesting TaskTracker as an input along with a set of unscheduled reduce tasks. For each reduce task that must be scheduled, the scheduler checks if the requesting TaskTracker is the one that contains the largest intermediate output. If so, the reduce task is scheduled on the requesting TaskTracker. Otherwise, another requesting TaskTracker is considered.

**Algorithm:**

Algorithm 1 Reduce Task Scheduler

<table>
<thead>
<tr>
<th>Input:</th>
<th>RT: set of unscheduled reduce tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TT: the task tracker requesting a reduce task</td>
</tr>
<tr>
<td></td>
<td>STT: set of task trackers</td>
</tr>
</tbody>
</table>

| Output: | A reduce task R ∈ RT that can be scheduled at TT or -1 |

1. for every reducer R ∈ RT do
2.  large = 0
3.  for every T ∈ STT do
4.      size = the size of the intermediate output on T
5.      if size > large then
6.          large = size
7.    end if
8.  end for
9. for end for
10. for the first reducer R ∈ RT do
11.    LTT = T
12.    if TT = LTT then
13.        return R
14.    end if
15. else
16.    return -1(Request with another task tracker)
17. end for

Identify which node has the maximum size intermediate output for each Reducer instance. Size of the intermediate output can be computed by referring the index file which is present on the JobTracker. Assign each Reducer to the TaskTracker which has maximum size intermediate output adequate amount of intermediate output. Multi-file count is another variation of word count that increases the workload and processes several files. Pi Estimator is suitable for testing the Hadoop scheduler when it deals with 1 KB data or lesser. It is a scenario with one reducer. This benchmark indicates that the modified Hadoop scheduler is capable of handling traffic prevalent within a single data center or a small cluster.

**RESULTS**

While running the MapReduce job, we submitted the job with different deadline values and estimated map cost, reduce cost and shuffle cost and kept the input size constant. The variation of map and reduce slots requirement is shown in Figure 5. We observed that resource demand will decrease as deadline is extended.

![Variation of Map slots and Reduce slots across different deadlines.](image)

To determine the effectiveness of resource optimizer, we ran workloads with IO intensive job, example - Random Writer, and CPU intensive jobs, example - pi estimator.

In our scenario, the duration of the IO-intensive job's and the CPU-intensive job's are kept nearly same and the two jobs are submitted together since it provides equal opportunity for fair scheduling as well as our implementation of the Hadoop scheduler in our Hadoop scheduling framework. There is only one job pool that consists of all the slots in the cluster as all jobs are submitted by the same user.

![Running time for jobs submitted with and without Resource Optimiser](image)
We used a mixed workload which involved a Random Writer job that generated 10GB data for every node and a Pi estimator job with 10,000,000 samples. The Pi estimator job had 3 map tasks and one reduce task and the Random Writer job had 30 map tasks and no reduce tasks. Results of our analysis are shown in Figure 5 and Figure 6. The data shuffled by scheduler was significantly lesser when we adopted the data locality feature. Our implementation of the Hadoop scheduler had approximately 7% more resource usage than the fair scheduler. According to our observations, on the fair scheduler, the map phase of Pi estimator took the whole jobs duration and the map phase of RandomWriter also took approximately its whole jobs duration, which implies that the workload had more opportunities to overlap complementary tasks. This could be explained by looking at the workloads, the Pi estimator is majorly a CPU-intensive job with barely any IO operation whereas the Random Writer is an extremely IO-intensive with CPU intensity [7].

Data locality feature mainly focuses on minimizing the transfer of intermediate output for further processing. The analysis results show a reduction of 11-40% in data traffic.

REFERENCES

CONCLUSION
Through this paper, we have proposed a New Hadoop scheduler framework that can be merged in to the future versions of Hadoop. This framework is a comprehensive model to make the scheduler more robust and quicker than it is presently.

Deadline scheduler module ensures there is no unwanted waiting for the job. Our observation shows that for a particular data size with increasing deadlines, resource demand will decrease. Also, if data size increases and deadline is kept constant, resource demand will increase. If resource demand increases, we can meet the demand by adding physical or virtual node to the existing cluster dynamically or provide a feasible deadline.

By maximizing the resource usage, we achieved an average increase of 7%-9% in a 4 node cluster. This was as achieved by overlapping CPU-Intensive tasks and IO-intensive tasks also taking care that fairness is not compromised.