Performance Improvement of MapReduce Framework in Heterogeneous Context using Reinforcement Learning

Nenavath Srinivas Naika, Atul Negi, V. N. Sastry

Abstract
MapReduce is presently established as an important distributed and parallel programming model with wide acclaim for large scale computing. Intelligent scheduling decisions can help in reducing the overall runtime of the jobs. MapReduce performance is currently limited by its default scheduler, which does not adapt well in heterogeneous environments. Heterogeneous environments were considered in Longest Approximate Time to End scheduler. This too has several shortcomings due to the static manner in which it computes progress of tasks. The lack of adequate approach to heterogeneous environments is currently being taken up in recent research. In this paper, we propose a novel MapReduce scheduler in heterogeneous environments based on Reinforcement learning called MapReduce Reinforcement Learning scheduler, which observes the system state of task execution and suggests speculative re-execution of the slower tasks to other available nodes in the cluster for faster execution. The proposed approach adapts to the heterogeneous environment and no prior knowledge of the environmental characteristics are required. It is expected that over a few runs the system would be able to better map the computing requirements to the resources available in a heterogeneous cluster and minimizes the overall job completion time.

Keywords: MapReduce; Reinforcement learning; Speculative execution; Job scheduler; Task scheduler.

1. Introduction
MapReduce is one of the popular computational frameworks for large-scale data processing and analysis for distributed computing and parallel processing systems. The structure of MapReduce is based on the master-slave architecture [1]. A single master node monitors the status of the slave nodes and assigns jobs to them. MapReduce schedulers are responsible for assigning the incoming tasks to available resources in the cluster [2]. However, there are various issues in MapReduce which may directly affects the performance of scheduler like heterogeneity, stragglers, number of jobs and resources. These issues have been undervalued.
As it has been widely adopted, improving the MapReduce performance is now a significant research topic. Reinforcement learning can solve a wide range of problems that are modeled as Markov or Semi-Markov Decision Processes and it has made a lot of attention from the research community. Reinforcement learning based scheduling algorithms will help the schedulers to make scheduling decisions efficiently. There are numerous successes of MapReduce approaches for machine learning and data mining problems but there is no significant research effort made on designing of the MapReduce scheduling algorithms with Reinforcement learning methods. By combining MapReduce parallel framework and Reinforcement learning algorithms will handles the complex scheduling decision problems more efficiently by maintaining the trade-off between exploring and exploiting. In this paper, the authors present a novel MapReduce scheduler with the Reinforcement learning approach for finding the straggler tasks efficiently in the Hadoop cluster which will improves the overall performance of the MapReduce framework.

The remainder of this paper is organized as follows. In Section II, overview of the MapReduce parallel programming model is presented. Background on MapReduce scheduling in heterogeneous environment is presented in Section III. Section IV presents the novel procedure for finding straggler tasks in MapReduce framework using Reinforcement learning approach. Finally, authors concludes the paper and presents some outlines of the future work in Section V.

2. Overview of MapReduce

The brief description of the MapReduce parallel programming model has been discussed in this section. It provides a simple map and reduce interfaces for users to specify the operations. The scheduling of MapReduce process contains following steps while scheduling a job from master to the slave nodes as illustrated in Fig. 1 [1].

- The Hadoop framework first breaks the input data into $M$ pieces of identical data size, which then are distributed in the cluster.
- The master node will pick up the idle worker nodes and assigns them $M$ map tasks. After intermediate output is produced by map tasks, the master node will assigns $R$ reduce tasks to the worker nodes which are idle.
- A worker node which is executing a map task parses the data block and feeds each (key, value) pair for the map function which are defined by the user. The intermediate (key, value) pairs from the map function are buffered in memory at the corresponding nodes that are executing them.
- The above buffered pairs are written to local disks at regular intervals and split into $R$ regions by (map) worker using a (configurable) partition function, default is $(hash$(intermediate key)) mod $R$, so that same intermediate (key, value) pairs go to one partition. When the map task is completes, worker sends the locations (file names) of partitions to the master.
- The master informs about the partition locations to the idle or running reducer workers. Then the reducer will read the data which is buffered from the local disks of map workers using remote procedure calls. After reading all the intermediate data, reducer worker sorts and groups the data by intermediate key so that all values of the same key are collected together.

3. MapReduce Scheduling in Heterogeneous Environment

The performance of MapReduce is primarily dependent on its task scheduler, minimizing the overall completion time of a job by appropriately assigning tasks to the available nodes is a common goal of the MapReduce scheduling. In Hadoop cluster, if a task is executing for longer period of time compared to other tasks then this condition will be called as straggler. Speculative tasks can cause resource wastage
and hamper other job performance in the cluster [12]. Even though MapReduce schedulers attempt to launch backup tasks for stragglers, they are failing to identify correct straggler tasks because of errors and difficulties in estimating the tasks remaining execution time [13]. This would lead to problems such as, launching a backup task for wrongly identified stragglers will not improve the MapReduce performance and the system resources are not utilized efficiently [15].

3.1. MapReduce default Scheduling algorithm

In MapReduce default scheduler [16], the progress score (PS) of a task $t$ is denoted by $PS_t$, which is calculated using (1) for map tasks and (2) for reduce tasks.

$$PS_t = \frac{M}{N}$$ (1)

$$PS_t = \frac{1}{3} (K + \frac{M}{N})$$ (2)

Where, $M$ is the number of (key, value) pairs that have been processed successfully, $N$ is the overall number of (key, value) pairs and $K$ is the stage (shuffle, sort and merge) value in a reduce phase.

The average progress score of a job $PS_{avg}$ is calculated using (3), $PS[i]$ is the progress score of a task $t_i$ and $n$ is the number of executable tasks in a job.

$$PS_{avg} = \frac{1}{n} \sum_{i=1}^{n} PS[i]/n$$ (3)

We can say that a task $t_i$ is a straggler task only if it satisfies (4), then launches the backup task for that particular straggler task $i$.

$$PS[i] < PS_{avg} - 20\%$$ (4)

3.1.1. The limitations of the above method are

- In MapReduce default scheduler, the map and reduce task weights are $M_1 = 1$, $M_2 = 0$ and $(R_1 = R_2 = R_3 = 1/3)$ but these weights will change when tasks run in a heterogeneous environment.

- Default scheduler can not identify the correct straggler tasks which need to be re-executed with fast nodes and sometimes the backup tasks will be launched for fast tasks instead of slow tasks.

- It uses difference in progress of 20% as threshold, it means that the tasks which has progress score above 80% will no longer speculatively executed as average progress score will never go beyond 100%. 

Fig. 1. MapReduce workflow
3.2. Longest Approximate Time to End MapReduce scheduling algorithm

M. Zaharia et al. [9] developed the Longest Approximate Time to End (LATE) scheduler for finding straggler tasks in heterogeneous environment. Progress rate of a task $t_j$ is $PR_j$, which is used to evaluate the remaining execution time of $t_j$ using (5) and $TTE_j$ denotes the remaining execution time of task $t_j$ and is evaluated using (6), where $T$ is the elapsed time.

$$PR_j = PS_j / T$$  \hspace{1cm} (5)

$$TTE_j = (1 - PS_j) / PR_j$$  \hspace{1cm} (6)

3.2.1. Advantages of LATE

- LATE primarily focuses on approximating the remaining execution time more willingly than just the progress score as it will speculatively executes only those tasks which will increase the overall job response time.
- LATE takes into account node heterogeneity when choosing a node to run a speculative task.

3.2.2. Limitations of LATE

- Even though LATE practices better approach to present backup tasks, it cannot always finds the actual straggler tasks since it does not approximate time to end of running tasks correctly.
- It uses same static approach as the MapReduce default scheduler for finding map and reduce stage weights.

4. Proposed MRRL (MapReduce Reinforcement Learning) based Scheduler

To address the scheduling issue of tasks for MapReduce in Heterogeneous nodes, authors uses a classical Reinforcement learning based algorithm called SARSA. Advantages of solving scheduling problems with Reinforcement learning are relatively easy modelling of the problem, it requires no prior knowledge of the environment dynamics and constructs fairly simple rewarding policy. The goal of the proposed scheduler is to find appropriate straggler tasks in a way to decrease the overall job completion time.

4.1. Parameters of the proposed MRRL scheduler

When Reinforcement learning method is applied to a scheduling problem, generally it need to define the state $S$ of the system, value function $V$ for each state, actions $A$ which can be taken, rules of transition $T$ between the states, model of the environment and the reward function $R$ which indicates the rewards received from the environment after each transition. The RL algorithm makes explorative and exploitative traverses in the state-space trying to find a path that is highly rewarded. In Q-learning, $Q$ (state-action) values can be learn on-line without a learning model of the environment.

4.1.1. Agent

Agent corresponds to a task of a job within a node of the cluster.

4.1.2. Environment

Environment is the world surrounding the agent.

4.1.3. State Space

In the proposed algorithm, the state representation consists parameters as Slow task, Fast task, Slow node and Fast node states. The task which has longer remaining execution time is the slow task state and a task which has smaller remaining execution time is the fast task state. The slow and fast nodes will be identified in the process of learning states of the environment. The node which has maximum number of negative rewards will be marked as slow node and which has maximum number of positive rewards will be marked as fast node.
4.1.4. Action Space

Action is the set of tasks that an agent can perform during each time period. Initially, the $\epsilon$-greedy method is adopted in this paper because of its simplicity and often ensures a sufficient exploitation and exploration balance. For each state in the environment, authors chooses its associated actions. The actions such as if each slow task in the node has to make an action then it has to launch backup task for that task which has longer remaining execution time from node $i$ to node $j$ in the cluster and there is no action for the fast task state.

4.1.5. Reward Function

The objective of a learning agent is described by the Reward function and it defines values of the instantaneous action built on the observed state of the environment. In this paper, authors defines different reward functions according to different system objectives and gives a positive reward for launching backup tasks for the slow tasks on the fast node and a negative reward for launching backup tasks for the slow tasks on the slow node in the cluster.

4.2. Representing $Q$-values in MRRL Scheduler

In this paper, authors uses CMAC (Cerebellar Model Articulation Controller) [14] hashing because keeping track of values will take huge amount of space and it will not generalize. By using CMAC, authors would like to minimize the requirement of space and generalization can be enabled at the same time. The CMAC will utilize the multiple tables with different hash functions and it allows $Q$ values for unknown (state,action) pairs which will be approximated by the $Q$ values of several nearby (state,actions) pairs. The values of nearby points can be hashed into the same bucket because it can affect the values of remaining nearby points and the values of faraway points will have no effect because they are hashed to different buckets.

4.3. SARSA algorithm in proposed MRRL scheduler

The authors uses SARSA (state $t$, action $t$, reward $t + 1$, state $t + 1$ and action $t + 1$) updates to the Q-learning algorithm which is an on-policy learning algorithm, where $Q$ value of the previous state is updated based on the reward and the action actually taken. The values of the look up table entries for each policy are calculated with the update of incremental step as in (7).

$$Q(s, a) = Q(s, a) + \alpha[r + \gamma(Q(s', a') - Q(s, a))] \quad (7)$$

Where, $Q(s, a)$ is the previous (state, action) value. The learning rate $\alpha \in (0, 1)$ determines the importance of the reward given by a policy over previous executions of the same policy. In this paper, authors likes to use the parameter $\alpha (= 0.10)$ which determines the learning rate. The discount factor $\gamma \in (0, 1)$ determines the importance of previous rewards of the same policy. Finally, the parameter $\gamma (= 0.95)$ is used as the discount factor.

The agent will choose which action to perform either considering past experiences (exploitation) or through trial of new actions (exploration). The exploration probability $\epsilon$ can be a constant (usually ranging between 0.1 and 0.5) or can be heuristically chosen by starting with a high value and then it gradually decreased. So, actions are chosen using the $\epsilon$-greedy method with parameter epsilon ($= 0.05$) to exploit learning 95% of the time but additionally do exploration by randomly choosing actions 5% of the time. The proposed MRRL algorithm will be invoked periodically to get the new state and reward.

**Algorithm 1** Proposed MRRL Algorithm

1: GetState and Reward (&NewState, &Reward)
2: NewAction $\leftarrow$ GetAction(&NewState)
3: PerformAction (&NewAction)
5: PrevState $\leftarrow$ NewState
6: PrevAction $\leftarrow$ NewAction
Algorithm 2 GetStateandReward Algorithm
1: if Task $i$ is a Map Task then
2: \hspace{1em} ProgressScore$_i$ $\leftarrow$ $M/N$
3: else
4: \hspace{1em} ProgressScore$_i$ $\leftarrow$ $1/3 \ast (K + M/N)$
5: end if
6: ProgressRate$_i$ $\leftarrow$ PS$_i$/ExecutionTime$_i$
7: TimetoEnd$_i$ $\leftarrow$ $(1 - PS_i)/\text{ProgressRate}_i$
8: NewState $\leftarrow$ TimetoEnd$_i$
9: TimetoEnd$_{avg}$ $\leftarrow$ $\sum_{i=1}^{T} \text{TimetoEnd}_i / T$
10: Reward $\leftarrow$ 0, Initially
11: if Task $i$ is a slow Task then
12: \hspace{1em} if slow task is moved to fast node then
13: \hspace{2em} Reward $\leftarrow +1$
14: \hspace{1em} else
15: \hspace{2em} Reward $\leftarrow -1$
16: \hspace{1em} end if
17: else
18: \hspace{1em} Do nothing
19: end if

Algorithm 3 GetAction Algorithm
1: if $(\text{rand}() < \epsilon)$ then
2: \hspace{1em} Action $\leftarrow$ RandomAction
3: else
4: \hspace{1em} Lookup Q tables for the NewState and choose action with max Q value
5: end if

Algorithm 4 PerformAction Algorithm
1: \hspace{1em} for each running task $i$ of the job do
2: \hspace{2em} if $(\text{TTE}_i - \text{TTE}_{avg}) > (\text{TTE}_{avg} \ast \text{Threshold})$ then
3: \hspace{3em} Move task $i$ to the fast node (Max Q value)
4: \hspace{2em} else
5: \hspace{3em} NoAction
6: \hspace{2em} end if
7: \hspace{1em} end for

Algorithm 5 MRRLSarsa Algorithm
1: $h_1 \leftarrow H_1(s,a)$
2: $h_{11} \leftarrow H_1(s_1,a_1)$
3: $h_2 \leftarrow H_2(s,a)$
4: $h_{22} \leftarrow H_2(s_1,a_1)$
5: $h_3 \leftarrow H_3(s,a)$
6: $h_{33} \leftarrow H_3(s_1,a_1)$
7: $Q_1[h_1] \leftarrow (1 - \alpha)Q_1[h_1] + \alpha(Reward + \gamma Q_1[h_{11}])$
8: $Q_2[h_2] \leftarrow (1 - \alpha)Q_2[h_2] + \alpha(Reward + \gamma Q_2[h_{22}])$
9: $Q_3[h_3] \leftarrow (1 - \alpha)Q_3[h_3] + \alpha(Reward + \gamma Q_3[h_{33}])$
10: $Q(s,a) \leftarrow Q_1[h_1] + Q_2[h_2] + Q_3[h_3]$
Where, \((s, a)\) are the old state-action values and \((s_1, a_1)\) are the new state-action values.

5. Conclusion and Future work

In this paper, authors proposed a novel MapReduce scheduling algorithm for Heterogeneous environment based on the Reinforcement learning approach, which will improve the performance of MapReduce framework by finding the accurate stragglers. Traditional MapReduce and LATE schedulers are not able to find the straggler tasks accurately because overall progress of a task leads to the wastage of system resources and authors presented them along with further discussion on their relative strengths and weaknesses. The authors used SARSA learning algorithm because it is a model free and solves most of the problems for searching optimal states whose state transition depends on a scheduler. As scheduling objective is to minimize the job completion time. Thus, in the proposed model, the state determination criterion and reward function are both built based on it. The advantage of proposed MRRL algorithm is that it requires no prior knowledge of the environmental characteristics.

As part of future research work, authors likes to incorporate more Reinforcement learning concepts into MapReduce scheduling to further improve the performance of MapReduce framework in Heterogeneous environments and extend these approaches to MapReduce scheduling in real world scenarios.

References