International Journal for Innovative Engineering and Management Research
A Peer Revieved Open Access International Journal
www.ijiemr.org

IJIEMR Transactions, online available on 15th Jan 2019. Link :
http://www.ijiemr.org/main/index.php?vol=Volume-08&issue=ISSUE-01

Title: SELF SLOT CONFIGURATION USING AWS NETWORKS FOR HADOOP CLUSTERS

Volume 08, Issue 01, Pages: 191–200.

Paper Authors
PILLARI BHARGAVI, P NAGESWARA RAO
Dept of CSE, Swetha Institute of Technology & Science, Tirupati, AP, India.

International Journal for Innovative Engineering and Management Research
A Peer Revieved Open Access International Journal
www.ijiemr.org

SELF SLOT CONFIGURATION USING AWS NETWORKS FOR HADOOP CLUSTERS

[1]PILLARI BHARGAVI, [2]P NAGESWARA RAO
Dept of CSE, Swetha Institute of Technology & Science, Tirupati, AP, India.
[1]lakshmidevi0417@gmail.com, [2]puttanr@rediffmail.com

Abstract— The Map Reduce gadget and its open authority execution Hadoop have became the real degree for adaptable investigation on expansive data sets as of late. One of the critical worries in Hadoop is the way to limit the achievement period (i.E., make span) of an arrangement of Map Reduce employments. The cutting-edge Hadoop simply permits static space design, i.E., settled quantities of guide openings and reduce openings all through the life of a collection. In any case, we discovered that any such static association would possibly spark off low framework asset uses and in addition lengthy completing length. Persuaded with the aid of this, we recommend sincere but effective plans which make use of space percentage in the middle of manual and decrease assignments as a tunable deal with for diminishing the makespan of a given set. By making use of the workload statistics of as of past due finished occupations, our plans step by step allots property (or openings) to outline reduce errands. We done the displayed plans in Hadoop V0.20.2 and assessed them with delegate MapReduce benchmarks at Amazon EC2. The trial results showcase the viability and power of our plans beneath both trustworthy workloads and more unpredictable combined workloads.

Index Terms—MapReduce jobs; Hadoop scheduling; reduced makespan; slot configuration;

1 INTRODUCTION

MapReduce [1] has turn out to be the leading paradigm in current years for parallel massive facts processing. Its open supply implementation Apache Hadoop [2] has additionally emerged as a famous platform for daily statistics processing and information evaluation. With the upward thrust of cloud computing, MapReduce is not just for internal information technique in massive groups. It is now convenient for a ordinary user to release a MapReduce cluster at the cloud, e.G., AWS MapReduce, for records-extensive applications.When increasingly packages are adopting the MapReduce framework, a way to enhance the overall performance of a MapReduce cluster becomes a focal point of research and improvement. Both academia and industry have positioned high-quality efforts on process scheduling, resource control, and Hadoop packages [3]–[11]. As a complicated system, Hadoop is configured with a huge set of machine parameters. While it gives the ability to customise the cluster for distinct programs, it's miles hard for users to apprehend and set the most useful values for those parameters. In this paper, we purpose to develop algorithms for adjusting a primary system parameter with the purpose to improve the overall performance (i.E., lessen the makespan) of a batch of MapReduce jobs.

A classic Hadoop cluster consists of a single grasp node and more than one slave nodes. The master node runs the JobTracker routine that's accountable for scheduling jobs and coordinating the execution of duties of every activity.

Each slave node runs the TaskTracker daemon for hosting the execution of MapReduce jobs. The idea of “slot” is used to signify the potential of accommodating responsibilities on every node. In a Hadoop machine, a slot is assigned as a map slot or a lessen slot serving map obligations or lessen duties, respectively. At any given time, simplest one project may be jogging in keeping with slot. The number of to be had slots in line with node certainly affords the most diploma of parallelization in Hadoop. Our experiments have shown that the slot configuration has a giant impact on machine overall performance. The Hadoop framework, however, makes use of fixed numbers of map slots and reduce slots at every node as the default putting in the course of the life of a cluster. The values on this fixed configuration are normally heuristic numbers with out considering activity traits.

Therefore, this static setting is not nicely optimized and may preclude the performance development of the whole cluster. In this work, we recommend and implement a new mechanism to dynamically allocate slots for map and reduce tasks. The primary goal of the brand new mechanism is to enhance the of completion time (i.E., the makespan) of a batch of MapReduce jobs whilst maintain the simplicity in implementation and control of the slot-primarily based Hadoop layout. The key idea of this new mechanism, named TuMM, is to automate the slot task ratio among map and reduce obligations in a cluster as a tunable knob for lowering the makespan of MapReduce jobs. The Workload Monitor (WM) and the Slot Assigner (SA) are

the two primary components brought by TuMM. The WM that resides in the JobTracker periodically collects the execution time data of these days completed responsibilities and estimates the prevailing map and reduce workloads inside the cluster. The SA module takes the estimation to determine and modify the slot ratio between map and reduce obligations for each slave node. With TuMM, the map and decrease phases of jobs may be better pipelined beneath precedence based schedulers, and for that reason the makespan is reduced. We further check out the dynamic slot assignments in heterogeneous environments, and advocate a new version of TuMM, named H TuMM, which sets the slot configurations for each person node to reduce the makespan of a batch of jobs. We implemented the offered schemes in Hadoop V0.20.2 and evaluated

them with representative MapReduce benchmarks at Amazon EC2. The experimental consequences show the effectiveness and robustness of our schemes below each simple workloads and more complex blended workloads. The rest of the paper is organized as follows. We give an explanation for the incentive of our work via some experimental examples in Section 2. We formulate the hassle and derive the most effective placing for static slot configuration in a homogeneous cluster in Section three. The layout info of the dynamic mechanism

International Journal for Innovative Engineering and Management Research
A Peer Revieved Open Access International Journal
www.ijiemr.org

for homogeneous clusters and heterogeneous clusters are presented in Section 4 and Section five.

2 MOTIVATION

Currently, the Hadoop framework uses fixed numbers of map slots and reduce slots on each node throughout the lifetime of a cluster. However, such a fixed slot configuration may lead to low resource utilizations and poor performance especially when the system is processing varying workloads. We here use two simple cases to exemplify this deficiency. In each case, three jobs are submitted to a Hadoop cluster with 4 slave nodes and each slave node has 4 available slots. Details of the experimental setup are introduced in Section 6. To illustrate the impact of resource assignments, we also consider different static settings for map and reduce slots on a slave node. For example, when the slot ratio is equal to 1:3, we have 1 map slot and 3 reduce slots available per node. We then measure the overall lengths (i.e., makespans) for processing a batch of jobs, which are shown in Fig. 1. Case 1: We first submit three Classification jobs to process a 10 GB movie rating data set. We observe that makespan is varying under different slot ratio settings and the best performance (i.e., shortest makespan) is achieved when each slave node has three map slots and one reduce slot, see the left column of Fig. 1.

To interpret this effect, we further plot the execution times of each task in Fig. 2. Clearly, Classification is a map-intensive application; for example, when we equally distribute resources (or slots) between map and reduce tasks, i.e., with the slot ratio of 2:2, the length of a map phase is longer than that of a reduce phase, see Fig. 2(a). It follows that each job's reduce phase (including shuffle operations and reduce operations) overlaps with its map phase for a long period.

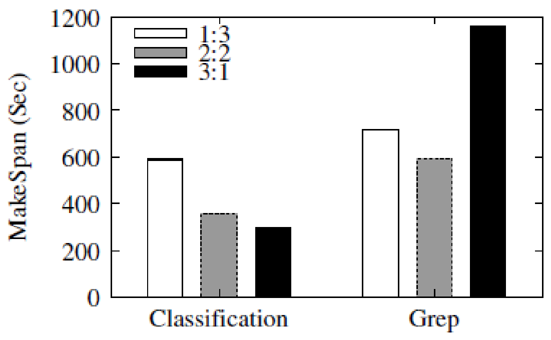
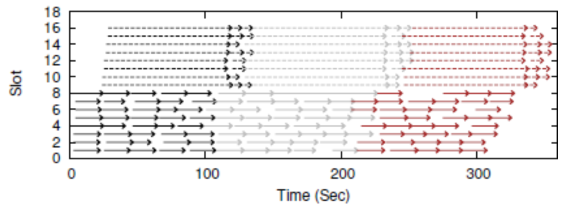
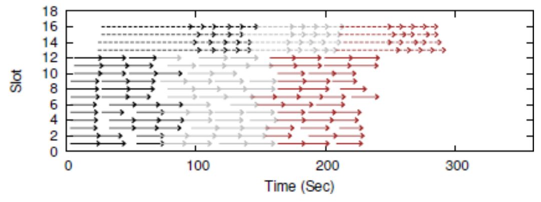


Fig. 1. The makespans of jobs under case 1 (i.e., Classification) and case 2 (i.e., Grep). The map and reduce slot ratios on each slave node are set to 1:3, 2:2, and 3:1.



(a) 2 map slots : 2 reduce slots



(b) 3 map slots : 1 reduce slot

Fig. 2. Task execution times of three Classification jobs under different static slot configurations, where each node has (a) 2 map slots and 2 reduce slots, and (b) 3 map slots and 1 reduce slot. Each arrowed line represents the execution of one task, and the solid (resp. dashed) ones represent map (resp. reduce) tasks. In addition, we use three different colors to discriminate the three jobs.

However, as the reduce operations can only start after the end of the map phase, the occupied reduce slots stay in shuffle for a long period, mainly waiting for the outputs from the map tasks. Consequently, system resources are underutilized. For example, we tracked the CPU utilizations of each task in a slave node every 5 seconds and Table 1 shows part of the records in one of such overlapping periods. At each moment, the overall CPU utilization (i.e., the summation of CPU utilizations of the four tasks) is much less than 400%, for a node with 4 cores. We then notice that when we assign more slots to map tasks, e.g., with the slot ratio of 3:1, each job experiences a shorter map phase and most of its reduce phase overlaps with the following job's map phase, see Fig. 2(b). The average CPU utilization is also increased by 20% compared to those under the the slot ratio of 2:2. It implies that for map-intensive jobs like Classification, one should assign more resources (slots) to map tasks in order to improve the performance in terms of makespan.

Case 2: In this case, we turn to consider reduceintensive applications by submitting three Grep jobs to scan the 10 GB movie rating data. Similar to case 1, we also investigate three static slot configurations.

TABLE 1

Real time CPU utilizations of each task on a slave node in the overlapping time period of a job's map and reduce phases. The slot ratio per node is 2:2.

Time(sec)	ProcessId/TaskType			
	3522/map	3564/map	3438/reduce	3397/reduce
1	14%	109%	26%	0%
6	103%	93%	0%	4%
11	93%	99%	8%	0%
16	100%	100%	0%	0%

First, we observe that each job takes a longer time to process its reduce phase than its map phase when we have 2 map and 2 reduce slots per node, see Fig. 3(a). Based on the observation in case 1, we expect a reduced makespan when assigning more slots to reduce tasks, e.g., with the slot ratio of 1:3. However, the experimental results show that the makespan under this slot ratio setting (1:3) becomes even longer than that under the setting of 2:2, see the right column of Fig. 1. We then look closely at the corresponding task execution times, see Fig. 3(b). We find that the reduce tasks indeed have excess slots such that the reduce phase of each job startstoo early and wastes time waiting for the output from its map phase. In fact, a good slot ratio should be setbetween 2:2 and 1:3 to enable each job's reduce phase to fully overlap with the following job's map phase rather than its own map phase.In summary, in order to reduce the makespan of a batch of jobs, more resources (or slots) should be assigned to map (resp. reduce) tasks if we have map (resp. reduce) intensive jobs. On the other hand, a simple adjustment in such slot configurations is not enough. An effective approach should tune the slot assignments such that the execution times of map and reduce phases can be well balanced and the makespan of a given set can be reduced to the end.

3 SYSTEM MODEL AND STATIC SLOT CONFIGURATION

In this section, we present a homogeneous Hadoop system model we considered and formulate the problem. In addition, we analyze the default static slot configuration in Hadoop and present an algorithm to derive the best configuration.

Problem Formulation

In our problem setting, we consider that a Hadoop cluster consisting of k nodes has received a batch of n jobs for processing. We use J to represent the set of jobs, J = f[j1; j2; : : : ; jn]. Each job ji is configured with nm(i) map tasks and nr(i) reduce tasks. Let st(i) and ft(i) indicate the start time and the finish timeof job ji, respectively. The total slots number in the Hadoop cluster is equal to S, and let sm and sr be the number of map slots and reduce slots, respectively. We then have S = sm + sr. In this paper, our objective is to develop an algorithm to dynamically tune the parameters of sm and sr, given a fixed value of S, in order to minimize the makespan of the given batch of jobs, i.e., minimizefmaxfft(i); 8i 2 [1; n]gg. Table 2 lists important notations that have been used in the rest of this paper.

TABLE 2

Notations used in this paper.

S, s_m, s_r	number of total/map/reduce slots of cluster;
$n_m(i), n_r(i)$	number of map/reduce tasks of job i;
$n'_m(i), n'_r(i)$	number of unscheduled map/reduce tasks of job i;
$\bar{t}_m(i), \bar{t}_r(i)$	average map/reduce task execution time of job i;
$w_m(i), w_r(i)$	total execution time of map/reduce tasks of job i;
$w'_m(i), w'_r(i)$	execution time of unscheduled tasks of job i;
$st(i), ft(i)$	start/finish time of job i;
rt_m, rt_r	number of currently running map/reduce tasks;

In a Hadoop system, makespan of multiple jobs also depends on the job scheduling algorithm which is coupled with our solution of allocating the map and reduce slots on each node. In this paper, we only consider using the default FIFO (First-In-First-Out) job scheduler because of the following two reasons. First, given n jobs waiting for service, the performance of FIFO is no worse than other schedulers in terms of makespan. In the example of "Case 2" mentioned in Section 2, the makespan under FIFO is 594 sec while Fair, an alternative scheduler in Hadoop, consumes 772 sec to finish jobs. Second, using FIFO simplifies the performance analysis because generally speaking, there are fewer concurrently running jobs at any time. Usually two jobs, with one in map phase and the other in reduce phase.

Furthermore, we use execution time to represent the workload of each job. As a MapReduce job is composed of two phases, we define wm(i) and wr(i) as the workload of map phase and reduce phase in job ji, respectively. We have developed solutions with and without the prior knowledge of the workload and we will discuss how to obtain this information later.

Static Slot Configuration withWorkload Information

First, we consider the scenario that the workload of a job is available and present the algorithm for staticslot configuration which is default in a Hadoop system.

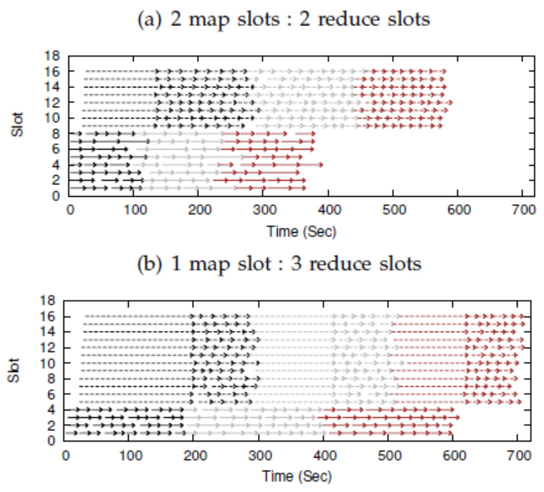


Fig. 3. Task execution times of a batch of Grep jobs under different static slot configurations, where each node has (a) 2

map slots and 2 reduce slots, and (b) 1 map slot and 3 reduce slots.

Basically, the Hadoop cluster preset the values of sm and sr under the constraint of S = sm + sr before executing the batch of jobs, and the slot assignment will not be changed during the entire process. We have developed the following Algorithm 1 to derive the optimal values of sm and sr.

Our algorithm and analysis are based on an observation that the time needed to finish the workload of map or reduce phase is inversely proportional to the number of slots assigned to the phase in a homogeneous Hadoop cluster. Given sm and sr, the map (resp. reduce) phase of ji needs nm(i)sm(resp. nr(i) sr) rounds to finish. In each round, sm map tasks or sr reduce tasks are processed in parallel and the time consumed is equal to the execution time of one map or one reduce task. Let tm(i) and tr(i) be the average execution time for a map task and a reduce task, respectively. The workloads of map and reduce phases are defined as

wm(i) = nm(i) _ tm(i);wr(i) = nr(i) _ tr(i): (1)

Algorithm 1 can derive the best static setting of sm and sr given the workload information. The outer loop (lines 1–10) in the algorithm enumerates the value of sm and sr (i.e., S□sm). For each setting of sm and sr, the algorithm first calculates the workload (wm(i) and wr(i)) for each job ji in lines 3–5. The second inner loop (lines 6–8) is to calculate the finish time of each job. Under the FIFO policy, there are at most two concurrently running jobs in the Hadoop cluster. Each job's map or reduce phase cannot start before the precedent job's map or reduce phase is finished. More specifically, the start time of map tasks of job ji, i.e., st(i), is the finish time of ji□1's map phase, i.e., st(i) = st(i . 1) + wm(i.1) sm . Additionally, the start time of ji's reduce phase should be no earlier than both the finish time of ji's map phase and the finish time of ji.1's reduce phase. Therefore, the finish time of ji is ft(i) = max(st(i) + wm(i) sm; ft(i . 1)) + wr(i) sr . Finally, the variables Opt SM and Opt MS keep track of the optimal value of sm and the corresponding makespan (lines 9–10), and the algorithm returns Opt SM and S Opt SM as the values for sm and sr at the end. The time complexity of the algorithm is O(S.n).

Algorithm 1 Static Slot Configuration

```
1:  for $s_m = 1$ to $S$ do
2:      $s_r = S - s_m$
3:      for $i = 1$ to $n$ do
4:          $w_m(i) = n_m(i) \cdot \bar{t}_m(i)$
5:          $w_r(i) = n_r(i) \cdot \bar{t}_r(i)$
6:      for $i = 1$ to $n$ do
7:          $st(i) = st(i-1) + \frac{w_m(i-1)}{s_m}$
8:          $ft(i) = \max(st(i) + \frac{w_m(i)}{s_m}, ft(i-1)) + \frac{w_r(i)}{s_r}$
9:      if $ft(n) < Opt\_MS$ then
10:         $Opt\_MS = ft(n); Opt\_SM = s_m$
11: return $Opt\_SM$ and $S - Opt\_SM$
```

4 DYNAMIC SLOT CONFIGURATION UNDER HOMOGENEOUS ENVIRONMENTS

As discussed in Section 2, the default Hadoop cluster uses static slot configuration and does not perform well for varying workloads. The inappropriate setting of sm and sr may lead to extra overhead because of the following two cases: (1) if job ji's map phase is completed later than job ji□1's reduce phase, then the reduce slots will be idle for the interval period of (st(i)+wm(i))□ft(i□1), see Fig. 4(a); (2) if job ji's map phase is completed earlier than the job ji□1's reduce phase, then ji's reduce

tasks have to wait for a period of ft(i □ 1) □ (st(i) + wm(i)) until reduce
slots are released by ji□1, see Fig. 4(b). In this section, we present our solutions that dynamically allocate the slots to map and reduce tasks during the execution of jobs. The architecture of our design is shown in Fig. 5. In dynamic slot configuration, when one slot becomes available upon the completion of a map or reduce task, the Hadoop system will reassign a map or reduce task to the slot based on the cPurrent optimal values of sm and sr. There are totally i2[1;n](nm(i) + nr(i)) tasks and at the end of each task, Hadoop needs to decide the role of the available slot (either a map slot or a reduce slot). In this setting, In this section, we present our solutions that dynamically allocate the slots to map and reduce tasks during the execution of jobs. The architecture of our design is shown in Fig. 5. In dynamic slot configuration, when one slot becomes available upon the completion of a map or reduce task, the Hadoop system will reassign a map or reduce task to the slot based on the cPurrent optimal values of sm and sr. There are totally i2[1;n](nm(i) + nr(i)) tasks and at the end of each task, Hadoop needs to decide the role of the available slot (either a map slot or a reduce slot). In this setting, therefore, we cannot enumerate all the possible values of sm and sr (i.e., 2 P I (nm(i)+nr(i)) combinations) as in Algorithm 1. Instead, we modify our objective in the dynamic slot configuration as there is no closed-form expression of the makespan.

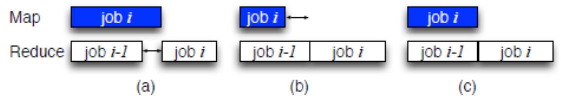


Fig. 4. Illustration of aligning the map and reduce phases. (a) and (b) are the two undesired cases mentioned above, and our goal is to achieve (c).Our goal now is, for the two concurrently running jobs (one in map phase and the other in reduce phase), to minimize the completion time of these two phases. Our intuition is to eliminate the two undesired cases mentioned above by aligning the completion of ji's map phase and ji□1's reduce phase, see Fig. 4(c). Briefly, we use the slot assignment as a tunable knob to change the level of parallelism of map or reduce tasks. When we assign more map slots, map tasks obtain more system resources and could be finished faster, and vice versa for reduce tasks. In the rest of this section, we first present our basic solution with the prior knowledge of job workload. Then, we describe how to estimate the workload in practice when it is not available. In addition, we present a feedback control-based solution to provide more accurate estimation of the workload. Finally, we discuss the design of task scheduler in compliance with our solution.

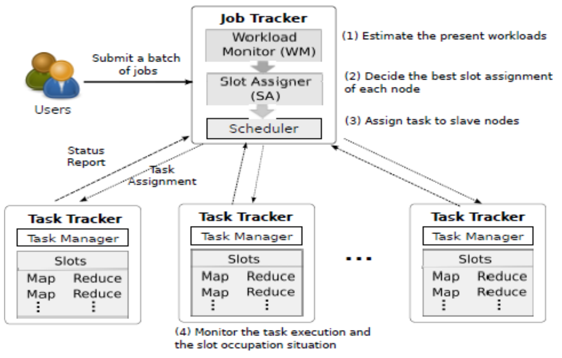

Job Tracker
Workload Monitor (WM)
Slot Assigner (SA)
Scheduler
Submit a batch of jobs
Users
(1) Estimate the present workloads
(2) Decide the best slot assignment of each node
(3) Assign task to slave nodes
Status Report
Task Assignment
Task Tracker
Task Manager
Slots
Map Reduce
Map Reduce
(4) Monitor the task execution and the slot occupation situation

Fig. 5. The architecture overview of our design. The shade rectangles indicate our new/modified components in Hadoop.

5 Workload Estimation

Our solution proposed above depends on prior knowledge of workload information. In practice, workload can be derived from job profiles, training phase, orother empirical settings. In some applications, however, workload information may not be available or accurate. In this subsection, we present a method that estimates the workload during the job execution without any prior knowledge.We use w0 m and w0 r to represent the remaining workload of a map or reduce phase, i.e., the summation of execution time of the unfinished map or reduce tasks. Note that we only track the map/reduce workloads of running jobs, but not the jobs waiting in the queue. Basically, the workload is calculated as the multiplication of the number of remaining tasks and the average task execution time of a job. Specifically, when a map or reduce task is finished, the current workload information needs to be updated, as shown in Algorithm 2, where n0 m(i)/ n0 r(i) is the number of unfinished map/reduce tasks of job ji, and tm(i)/ tr(i) means the average execution time of finished map/reduce tasks from ji. Note that the execution time of each finished task is already collected and reported to the JobTracker in current Hadoop systems. In addition, we use the Welford's one pass algorithm to calculate the average of task execution times, which incurs very low overheads on both time and memory space.

Algorithm 2 Workload Information Collector

```
if a map task of job $j_i$ is finished then
    update the average execution time of a map task $\bar{t}_m(i)$
    $w'_m(i) = \bar{t}_m(i) \cdot n'_m(i)$
if a reduce task of job $j_i$ is finished then
    update the average execution time of a reduce task $\bar{t}_r(i)$
    $w'_r(i) = \bar{t}_r(i) \cdot n'_r(i)$
```

6 Slot Assigner

The task assignment in Hadoop works in a heartbeat fashion: the TaskTrackers report slots occupation situation to the JobTracker with heartbeat messages; and the JobTracker selects tasks from the queue and assigns them to free slots. There are two new problems need to be addressed when assigning tasks under TuMM. First, slots of each type should be evenly distributed across the slave nodes. For example, when we have a new slot assignment sm = 5; sr = 7 in a cluster with 2 slave nodes, a 2:3/4:3 map/reduce slots distribution is better than the 1:4/5:2 map/reduce slots distribution in case of resource contention. Second, the currently running tasks may stick with their slots and therefore the new slot assignments may not be able to apply immediately.

To address these problems, our slot assignment module (SA) takes both the slots assignment calculated through Eq. 6-7 and the situation of currently running tasks into consideration when assigning tasks.

The process of SA is shown in Algorithm 3. The SA first calculates the map and reduce slot assignments of slave node x (line 1), indicated by sm(x) and sr(x), based on the current values of sm and sr and the number of running tasks in cluster. We use the floor function since slots assignments on each node must be integers. Due to the flooring operation, the assigned slots (sm(x)+sr(x)) on node x may be fewer than the available slots (S=k). In lines 3–6, we increase either sm(x) or sr(x) to compensate slot assignment. The decision is based on the deficit of current map and reduce slots (line 3), where sm/ sr represent our target assignment and rtm/ rtr are the number of current running map/reduce tasks.

Eventually, we assign a task to the available slot in lines 7–10. Similarly, the decision is made by comparing the deficit of map and reduce tasks on node x, where sm(x)/ sr(x) are our target assignment and rtm(x)/ rtr(x) are the numbers of running tasks.

Algorithm 3 Slot Assigner

```
0:  Input: Number of slave nodes in cluster: $k$
    Total numbers of running map/reduce tasks: $rt_m$, $rt_r$;
0:  When receive heartbeat message from node $x$ with the
    number of running map/reduce tasks on node $x$: $rt_m(x)$,
    $rt_r(x)$;
1:  Initialize assignment of slots for node $x$:
    $s_m(x) \leftarrow \lfloor s_m/k \rfloor, s_r(x) \leftarrow \lfloor s_r/k \rfloor$;
2:  if $(s_m(x) + s_r(x)) < S/k$ then
3:      if $(s_m - rt_m) > (s_r - rt_r)$ then
4:          $s_m(x) \leftarrow s_m(x) + 1$;
5:      else
6:          $s_r(x) \leftarrow s_r(x) + 1$;
7:  if $(s_m(x) - rt_m(x)) > (s_r(x) - rt_r(x))$ then
8:      assign a map task to node $x$;
9:  else
10:     assign a reduce task to node $x$.
```

7 CONCLUSION

In this paper, we supplied a unique slot management scheme, named TuMM, to allow dynamic slot configuration in Hadoop. The principal goal of TuMM is to improve aid utilization and decrease the makespan of more than one jobs. To meet this goal, the provided scheme introduces important components: Workload Monitor periodically tracks the execution facts of lately completed obligations and estimates the present workloads of map and reduce tasks and Slot Assigner dynamically allocates the slots to map and decrease responsibilities with the aid of leveraging the anticipated workload statistics. We similarly prolonged our scheme to manipulate assets (slots) for heterogeneous clusters. The new edition of our scheme, named H TuMM, reduces the makespan of a couple of jobs by means of one at a time placing the slot assignments for the node in a heterogeneous cluster. We carried out TuMM and H TuMM on the top of Hadoop v0.20.2 and evaluated each schemes with the aid of going for walks representative MapReduce benchmarks and TPC-H question units in Amazon EC2 clusters. The experimental effects exhibit up to twenty-eight% reduction in the makespans and 20% increase in resource utilizations. The effectiveness and the robustness of our new slot control schemes are demonstrated under both homogeneous and heterogeneous cluster environments.

In the future, we are able to similarly investigate the most fulfilling overall slot quantity configuration within the slot based totally Hadoop platform in addition to the useful resource control policy in subsequent generation Hadoop YARN structures.

Reference

[1].Apache Hadoop. [Online]. Available: http://hadoop.apache.org/

[2].M. Zaharia, D. Borthakur, J. S. Sarma et al., “Delay scheduling:A simple technique for achieving locality and fairness in cluster scheduling,” in EuroSys’10, 2010.

[3].A. Verma, L. Cherkasova, and R. H. Campbell, “Two sides of a coin: Optimizing the schedule of mapreduce jobs to minimize their makespan and improve cluster performance,” in MASCOTS’ 12, Aug 2012.

[4].M. Isard, Vijayan Prabhakaran, J. Currey et al., “Quincy: fairscheduling for distributed computing clusters,” in SOSP’09, 2009,pp. 261–276.

[5]. A. Verma, Ludmila Cherkasova, and R. H. Campbell, “Aria:Automatic

resource inference and allocation for mapreduce environments,"in ICAC'11, 2011, pp. 235–244.

[6]. J. Polo, D. Carrera, Y. Becerra et al., "Performance-driven task coschedulingfor mapreduce environments," in NOMS'10, 2010.

[7].V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar,R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth et al., "Apachehadoop yarn: Yet another resource negotiator," in Proceedings ofthe 4th annual Symposium on Cloud Computing. ACM, 2013, p. 5.

[8].X. W. Wang, J. Zhang, H. M. Liao, and L. Zha, "Dynamic splitmodel of resource utilization in mapreduce," in DataCloud-SC '11,2011.