International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2013): 6.14 | Impact Factor (2014): 5.611

PRISM: Phase and Resource Information-Aware Scheduler for MapReduce

P Ramesh Naidu[1], Dr. Guruprasad[2]

[1]Department of Computer Science and Engineering, Sri Venkateshwara College of Engineering, Bengaluru

[2]Department of Computer Science and Engineering, New Horizon College of Engineering, Bengaluru

Abstract: ***MapReduce is one of the important concepts of Hadoop that is used for data handling used by big companies today such as Google and Facebook. Here we divide each job into the map and reduce phases and try to complete the execution of the assigned task in a parallel form. In this paper, we suggest that it would be more efficient if we make the scheduler to work at the phase-level instead of the task-level. The reason is because the task demands a lot of requirements during its lifetime. For this very purpose, we introduce the concept called PRISM, which is aphase and information-aware scheduler for MapReduce and in this concept we divide the tasks into unequal parts called as phases and apply phase-level scheduling to these phases and achieve efficient resource usage***

Keywords: MapReduce, Hadoop, scheduling, resource allocation

1. Introduction

Businesses today are increasingly reliant on large scale data analytics to make critical day-to-day business decisions. This shift towards data-driven decision making has fueled the development of MapReduce [10], a parallel programming model that has become synonymous with large-scale, data-intensive computation. In MapReduce, a job is a collection of *Map* and *Reduce* tasks that can be scheduled concurrently on multiple machines, resulting in significant reduction in job running time. Many large companies, such as Google, Facebook, and Yahoo!, use only MapReduce to process large volumes of data on a daily basis. Consequently, the performance and efficiency of MapReduce is good, but not that great.

A central component of a MapReduce system is its job scheduler. The role of the job scheduler is to create a schedule of the Map and Reduce tasks that minimizes job completion time and maximizes resource utilization. When we apply a schedule with many tasks to a single machine, then there will be resource contention and takes more time to complete the job. Conversely, a schedule with too few concurrently running tasks on a single machine will cause the machine to have poor resource utilization.

The job scheduling problem can be easier to solve if we can assume that all map and reduce tasks have homogenous resource requirements in terms of CPU, memory, disk and network bandwidth. Indeed, current MapReduce systems, such as Hadoop MapReduce Version 1. x, make this assumption to simplify the scheduling problem. Unfortunately, in practice, run-time resource consumption varies from task to task and from job to job. Several recent studies have reported that production workloads often have diverse utilization profiles and performance requirements [8]. Failing to consider these job usage characteristics can potentially lead to inefficient job schedules with low resource utilization and long job execution time. Due to this disadvantage RAS i.e. resource aware scheduling and Hadoop MapReduce Version 2 [7], have introduced resource-aware job schedulers to the MapReduce framework. However, these schedulers specify a fixed size for each task in terms of required resources (e. g. CPU and memory), thus assuming the run-time resource consumption of the task is stable over its lifetime. In particular, it has been reported that the execution of each MapReduce task can be divided into multiple phases of data transfer, processing and storage [11]. A *task is divided into small unequal sizes called phases*. The phases involved in the same task can have different resource demand in terms of CPU, memory, disk and network usage. Therefore, scheduled tasks based on fixed resource requirements over their durations will often cause either excessive resource contention by scheduling too many simultaneous tasks on a machine.

In this paper, we present PRISM, a **P**hase and **R**esource **I**nformation-aware **S**cheduler for **M**apReduce clusters that performs resource-aware scheduling at the phase level. Therefore, by initial finding out the resource demand at the phase level, it is possible for the scheduler to maintain parallelism and at the same time avoiding resource contention. We suggest a phase–level scheduling algorithm for this and show that PRISM produces up to 18% improvement in resource utilization while allowing jobs to complete up to 1.3*times* faster than current Hadoop schedulers.

2. Existing System

2.1 Hadoop MapReduce

MapReduce [10] is a parallel computing model for large-scale data-intensive computations. A MapReduce job consists of two types of tasks, i.e. the map task and the reduce task. A map task takes a keyvalue block as the input that is stored in the underlying distributed file system and runs a user-specified map function to of key-value output. Subsequently, a reduce task is responsible for collecting and applying specified reduce function on the collected key value pairs to produce the final output. Currently, the most popular implementation of MapReduce is Apache Hadoop

MapReduce [1]. A Hadoop cluster consists of a collection of machines where one node will act as a master node and all the remaining n-1 nodes act as the slave node. The slave nodes execute the tasks assigned by the master node. The master node runs a resource manager (also known as a job tracker) that is responsible for scheduling tasks on slave nodes. Each slave node runs a local node manager (also known as a task tracker) that is responsible for launching and allocating resources for each task. To do so, the task tracker launches a Java Virtual Machine (JVM) that executes the corresponding map or reduce task. The original Hadoop MapReduce (i.e. version 1.*x* and earlier) adopts a slot-based resource allocation scheme. The scheduler assigns tasks to be executed to each machine based on the availability of the resources on each machine. The number of map and educe slots determine how the data are divided and allocated to each machine. As a Hadoop cluster is usually a multi-user system, many users can simultaneously submit jobs to the cluster. The job scheduling is performed by the resource manager in the master node, which maintains a list of jobs in the system. Here each slave node performs a small job and informs its completion via a heartbeat message (usually between 1-3 seconds) to the master node. The resource scheduler will use the provided information to make scheduling decisions. Today there are two commonly used schedulers that are: Capacity scheduler [2] and Fair scheduler [3]. These schedulers function on at task level.

2.2 MapReduce Job Phases

Current Hadoop job schedulers perform as task-level scheduling where initially a task given by the user to execute is divided into blocks or chunks which are of unequal size this is the map phase. In particular, a map task can be divided into 2 main phases: *map* and *merge*2. The Hadoop Distributed File System (HDFS) [4], where data blocks are stored across multiple slave nodes. In the map phase, a mapper fetches an input data block from the Hadoop Distributed File System (HDFS) [4] and applies the user - as with the Hadoop implementation, defined a map function on each record. The map function generates records that are serialized and collected into a buffer. When the buffer becomes full (i.e., content size exceeds a pre-specified threshold), the content of the buffer will be written to the local disk. Lastly, the mapper executors a merge phase to group the output records based on the intermediary keys, and store the records in multiple files so that each file can be fetched a corresponding reducer. Similarly, the execution of a reduce task can be divided into 3 phases: *shuffle*, *sort*, and *reduce*. In the shuffle phase, the reducer fetches the output file from the local storage of each map task and then places it in a storage buffer that can be either in memory or on disk depending on the size of the content. At the same time, the reducer also launches one or more threads to perform local merge sort in order to reduce the running time of the subsequent sort phase. Once all the map output records have been collected, the sort phase will perform one final sorting procedure to ensure all collected records are in order. Finally, in 1. Other resources such as disk and network I/O are yet to be supported by Hadoop Yarn.
2. We use the same phase names

3. Phase-Level Resource Requirements

Here we analyze the run-time resource requirements in each phase for various jobs that belong to Hadoop. We use Apache Hadoop 0.202 which is run using a 16 node environment where one node acts as master node and the remaining 15 node acts as slaves. Each node uses a quad core CPU with 12GB memory and 1TB local disk storage.

Here we evaluate the phase-level resource requirements across various jobs. The CPU and memory usage of each phase are collected using the Linux command called top and the input-output usage are collected by reading MapReduce I/O Counters at runtime.

We would actually prefer to divide certain phases into still finer portions to achieve even more uniform resource usage, but there may be system complexity involved and the scheduling overhead may outweigh the gain attended by phase-level scheduling.

3.1 Proposed System (PRISM)

It is said if the resource allocated to a machine is insufficient then it will affect the performance because time will be taken to complete execution of a task. This motivates us to design a fine-grained, phase-level scheduling scheme that allocates resources according to the phase that each task is currently executing. By exploiting fine-grained phase-level resource characteristics, it is possible to better "bin-pack" tasks on machines to achieve higher resource utilization compared to task-level schedulers.

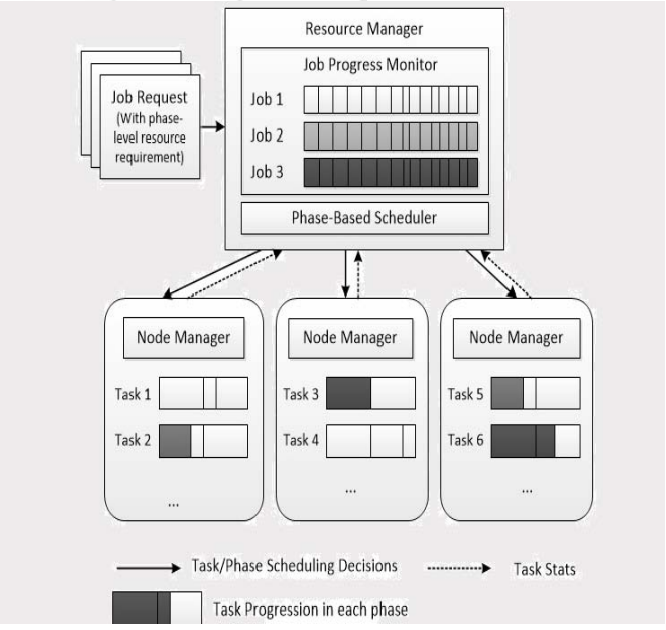

Figure 1: System Architecture.

Here in the proposed system we present the PRISM, a fine-grained resource-aware scheduler, which performs scheduling at phase-level. PRISM allows the job owners to specify the phase-level requirements. The above figure gives the description about the system architecture. The architecture states that there are majorly three components: a

scheduler called the phase-based scheduler which is located at the master node, local node managers that coordinate phase transitions with the scheduler and a job progress monitor that is indeed used progress information at the phase-level. The below figure shows phase-level scheduling mechanism that explains a series of actions that takes place within this architecture. First, whenever a task needs to be scheduled, the scheduler replies with a heart beat message with the task scheduling request. Then the node manager then assigns the task. Each time a task finishes executing a phase it notifies and asks permission of the node manager to go to the next phase. The node manager than forwards the permission request to the scheduler through the regular heartbeat message [10].If sufficient resources are available the scheduler decides and informs its decision to the local node manager whether it can proceed or wait the execution of the next phase. Finally, if the task is given permission to execute the next phase, the node manager grants the task to continue its duty. Once the task is completed, the task status is forwarded to the node manager and then forwarded again to the scheduler

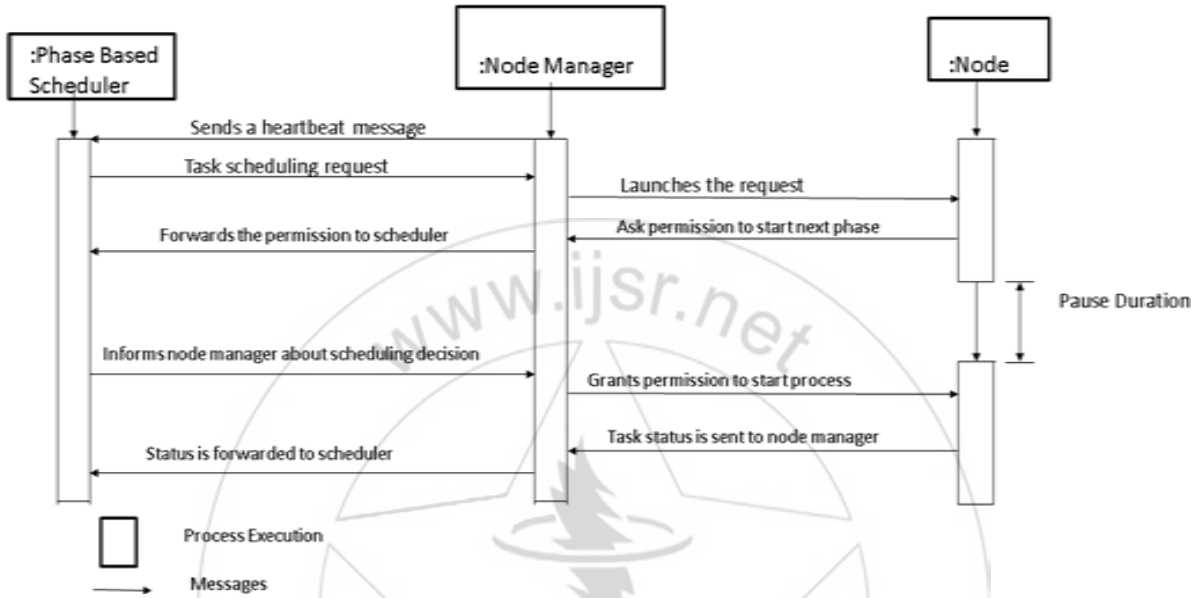

Figure 2: Phase-level scheduling mechanism

4. Scheduler Design

In this section, we describe in detail the design of PRISM's phase-based scheduling algorithm.

4.1 Design

The responsibility of a MapReduce job scheduler is to assign tasks to machines with consideration for both efficiency and fairness [8]. Efficiency can be achieved on proper resource allocation. Job running time is another parameter for resource efficiency because if the job is being able to compete is a shorter time then we can say that our machine runs efficiently. In contrast, fairness ensures that resources are fairly divided among jobs such that no job will experience starvation due to unfair resource allocation. However, simultaneously achieving both fairness and efficiency is quite difficult [10].Fair is scheduled algorithms such as Hadoop Fair Scheduler [3], Quincy [11] and Dominant Resource Fairness (DRF) [11] generally runs an iterative procedure by identifying users that experience the highest degree of unfairness (i.e. deficit) in each iteration, and schedule tasks that belong to those users to improve the overall fairness of the system. However, directly applying a fair scheduling algorithm for phase level scheduling is insufficient. In particular, given a set of phases that can be scheduled on a machine, the scheduling algorithm must consider their interdependencies in addition to their resource requirements . In many cases, such delays can also propagate two phases in other tasks, causing them to be delayed as well. For example, even though the execution of a shuffle phase of a reduce task can overlap with the execution of a merge phase of a map task, the shuffle phase cannot finish unless all merge phases of the map tasks have finished. Thus, when choosing between scheduling merge phase and shuffle phases, it is preferable to give sufficient resources to merge phases to allow them to finish faster, instead of allocating most of the resources to the shuffle phase and delay the completion of merge phases.

4.2 Algorithm description

We formally introduce our scheduling algorithm in this section. Upon receiving a heartbeat message from a node manager reporting resource availability on the node, the scheduler must select which phase should be scheduled on the node. Suppose there are J jobs and in the system. Specifically, each job j 2 J consists of two types of tasks: map tasks M and reduce task R. Let τ (t) 2 fM,Rg denote the type of a task t. Given a phase i of a task t that can be scheduled on a machine n, we define the utility function of assigning a phase i to machine n as:

$$U\,(i,\,n) = Unfairness\,(i,\,n) + \alpha\ \_Uperf\,(i,\,n) \qquad (1)$$

Where *Unfairness* and *Uperf* represent the utilities for improving fairness and job performance, respectively, and α

is an adjustable weight factor. If we set α close to zero, then the algorithm would greedily schedule phases according to the improvement in fairness. Notice that considering job performance objectives will not severely hurt fairness. When a job is severely below its fair share, scheduling any phase with non-zero resource requirement will only improve its fairness. Now we describe each term in Eq. (1). We define

$$Unfairness(i,n)=Ubeforefairness(i,n)+UafterFairness(i,n) \quad (2)$$

Where *Ubeforefairness* (*i,n*) and *Uafterfairness*(*i, n*) are the fairness measures of the job before and after scheduling.

Algorithm 1 Phase-Level Scheduling Algorithm

1: Upon receiving a status message from machine *n*:
2: Obtain the resource utilization of machine *n*
3: *PhaseSelected* ← {∅}
4: *Candidatep hases* ← {∅}
5: **repeat**
6: **for** each job *j* ∈ *jobsthathastasksonn* **do**
7: **for** each scheduable phase *i* ∈ *j* **do**
8:*CandidatePhases*←*CandidatePhases* ∪ {*i*}
9: **end for**
10: **end for**
11: **for** each job *j* ∈ top *k* jobs with highest deficit *n* **do**
12: **if** exist schedulable data local task **then**
13:*CandidatePhases*←*CandidatePhases* ∪{first phase of the local task *i*}
14: **else**
15:*CandidatePhases*←*CandidatePhases* ∪ {first phase of the non-local task *i*}
16: **end if**
17: **end for**
18: **if** *Candidate Phases* ≠ *null* **then**
19: **for** *i* ∈ *Candidate Phases* **do**
20: **if** *i* is not schedulable on *n* given current utilization **then**
21:*CandidatePhases*←*CandidatePhases*\{*i*}
22: continue;
23: **end if**
24: Compute the utility *U*(*i; n*) as in equation (1)
25: **if** *U*(*i; n*) ≤ 0 **then**
26:*CandidatePhases*←*CandidatePhases*\{*i*}
27: **end if**
28: **end for**
29: **if** *Candidate Phases* ≠ *NULL* **then**
30: *i* ← task with highest *U*(*i; n*) in the *Candidate Phases*
31: *PhaseSelected* ← *PhaseSelected* ∪ {*i*}
32:*CandidatePhases*←*CandidatePhases*\{*i*}
33: Update the resource utilization of machine *n*
34: **end if**
35: **end if**
36: **until** *Candidate Phases* == *NULL*
37: return *PhaseSelected*

5. Conclusion

MapReduce is a famous and important programming concept used for computing large data. Although there are many schedulers existing today that are resource-efficient our proposed work which is PRISM A fine-grained resource-aware scheduler that coordinates task execution at the task execution at the level of phases. Here we first demonstrate how different the task run-time over a variety of MapReduce jobs. We introduce a phase-level scheduling algorithm that is said to improve the job execution without introducing stragglers. In a 16 node Hadoop cluster running standard benchmarks, we show that PRISM provides high resource utilization and provides 1.3x improvement in job running time compares to the existing Hadoop schedulers.

References

[1] Hadoop MapReduce distribution. http://hadoop .apache. org.
[2] Hadoop Capacity Scheduler, http://hadoop.apache.org/docs/Stable/capacity scheduler.html/.
[3] Hadoop Fair Scheduler. http://hadoop.apache.org/docs/r0.20.2/fair scheduler.html.
[4] Hadoop Distributed File System,hadoop.apache .org/docs/hdfs/current/.
[5] GridMix benchmark for Hadoop clusters. http://hadoop.apache.org/docs/mapreduce/current/gridmi x. html.
[6] PUMABenchmarks,http://web.ics.purdue.edu/fahmad /benchmarks/datasets.htm.
[7] The Next Generation of Apache Hadoop MapReduce. http://hadoop.apache.org/docs/current/hadoop-yarn/
[8] R. Boutaba, L. Cheng, and Q. Zhang. On cloud computational models and the heterogeneity challenge. *Journal of Internet Services and Applications*, pages 1–10, 2012.
[9] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.
[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[11] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

Author Profile

P Ramesh Naidu received the B.E and M.Tech degree in Computer Science and Engineering from JNTU, Hyderabad. He is having a work experience of 10 years in Teaching and 1year in Industry. Published 4 international Journals and 2 National papers. SCJP Certified by Sun Microsystems. He is also member of professional societies like MIE and ISTE

Dr. N Guruprasad is basically a graduate, post graduate and doctorate from the field of Computer Science having 23 years of teaching experience. He is currently working as Professor in Computer Science Department at New Horizon college of Engineering.

To his credit he has more than 30 publications both at National and at International level. He has also authored books on C Programming and Data Structures. He is also member of professional societies like CSI, IETE, ISTE and ORSI