15-440: Project 4

Maximizing Data Locality in Hadoop Clusters via Controlled Reduce Task Scheduling

Project posted on: November 19, 2011

Interim design report: Not required

Final due date: December 13, 2011

I- Intended Learning Outcomes (ILOs)

This project applies task scheduling in Hadoop MapReduce. The ILOs of the project are:

1. Understand scheduling in distributed systems.
2. Apply data locality to task scheduling in Hadoop, a widely adopted large-scale data-intensive computing platform.
3. Empirically analyze Hadoop's performance and network traffic.

II- Project Objectives

The overall goal of this project is to gain a hands-on experience with working on a large open-ended research-oriented project using the Hadoop framework. Hadoop is an open source implementation of MapReduce and Google File System, and is currently enjoying wide popularity. Students will modify the task scheduler of Hadoop, conduct several experimental studies, and analyze performance and network traffic results.

The study of scheduling algorithms for distributed computing resources has a long history. However, there has been a renewed interest in the subject since the inception of MapReduce on the computing stage. The deployment of MapReduce by industry's premier web vendors- Facebook, Google, Amazon and Yahoo!- has, in fact, thrust MapReduce to the forefront of data-intensive computing.

A main challenge in distributed scheduling is deciding upon the location of data relative to the available computational resources. If computations are not placed close to their input data, the cluster network can become a bottleneck. When parallel tasks expose high cross-cluster network traffic, they start competing for the available network bandwidth. As such, optimizing the placement of computation to minimize network traffic is typically a primary goal of a data-intensive computing platform. You will have the chance to apply data locality to reduce task scheduling in Hadoop MapReduce. The project is aimed to provide you with a practical experience on such a realistic distributed computing platform as well as with a thorough treatment of an important problem in distributed systems (i.e., distributed scheduling).

III- Background on Hadoop

MapReduce is a programming model designed for processing sheer volumes of data concurrently by dividing the work into a set of independent tasks. Hadoop is an open source implementation of MapReduce. Hadoop MapReduce assumes a master-slave architecture. The master is referred to as Job Tracker (JT) and each slave is referred to as Task Tracker (TT). In addition to MapReduce, Hadoop employs its own distributed file system called Hadoop Distributed File System (HDFS). HDFS is designed to hold very large amount of data (TBs or even PBs) and provide high throughput data access. Hadoop MapReduce uses HDFS as an underlying storage layer to store application data sets.

In Hadoop MapReduce, an application is represented as a job. A job encompasses multiple Map and Reduce tasks. The input data to a job is divided into fixed-size pieces denoted as splits. HDFS reliably stores petabytes of data on clusters of commodity hardware via *replicating* splits on different nodes (by default, 3 replicas per split in Hadoop).

To process splits concurrently, each split is assigned a Map task by JT. A Map task processes an input split and produces intermediate key-value pairs that are partitioned/hashed to either zero (in case of map-only jobs) or many Reducers. A Reducer can collect one or many intermediate outputs from one or many feeding mappers located at one or many cluster nodes in a process typically known as ***shuffling***. Each reducer then merges and processes its inputs from mappers.

IV- Hadoop Network Toplogy

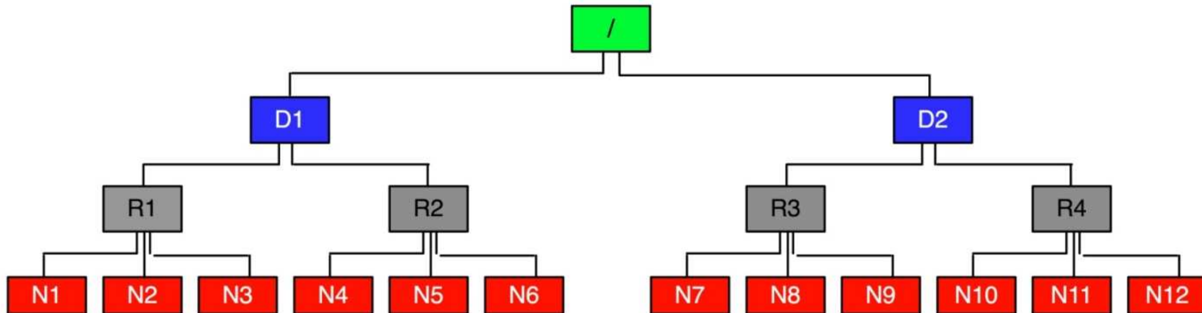

Figure 1: Network Topology in Hadoop (D= data center, R = rack, and N = node)

Hadoop assumes a tree-style network topology similar to the one exhibited in Fig. 1. Nodes are spread over different racks encompassed in one or many data centers. A salient point is that the bandwidth between two nodes is dependent on their relative locations in the network topology. For example, nodes that are on the same rack will have higher bandwidth between them as opposed to nodes that are off-rack. Rather than measuring bandwidth between two nodes, which might be difficult in practice, Hadoop adopts a simple approach via:

1- Representing a node's network position by a string (e.g., N9's location in Fig. 1 is represented by /D2/R3/N9).
2- Representing the bandwidth between two nodes as a measure of distance.
3- Assuming the distance from a node to its parent is 1.
4- Calculating the distance between any two nodes by adding up their distances to their closest common ancestor (e.g., ***Distance***(/D1/R1/N3, /D2/R4/N10) = 6).

V- Hadoop Heartbeat Protocol

The Job Tracker (JT) and Task Trackers (TTs) communicate over a Hadoop cluster network via a *periodic* heartbeat mechanism. In essence the heartbeat is a mechanism for a TT to announce its availability to JT. Besides, the heartbeat is used by TTs to send information about their states to JT (e.g., a heartbeat message can indicate whether a

TT is ready for a new task to run or not). JT processes the status information sent by TTs and responds with instructions to start/stop tasks or jobs, and also reset instructions during contingencies.

VI- Task Scheduling In Hadoop

Hadoop's framework adopts a *pull* scheduling strategy rather than a push one. That is, JT does not push map and reduce tasks to TTs but rather TTs pull them by making pertaining requests. Every TT sends a heartbeat message periodically to JT encompassing a request for a Map or a Reduce task to run. JT satisfies requests for Map tasks via attempting to schedule Mappers in the ***vicinity*** of their input splits in order to diminish network traffic. Hence, Hadoop MapReduce is said to exploit ***data locality*** when scheduling Map tasks. On the other hand, JT simply assigns the next yet-to-run Reduce task to a requesting TT regardless of TT's network location and its implied effect on the network traffic. As such, Hadoop MapReduce is said not to consider data locality when scheduling reduce tasks.

VII- The Problem in Hadoop's Task Scheduling

Let us explain the problem incurred by Hadoop as a result of not considering data locality when scheduling Reduce tasks. Fig. 2 demonstrates a simple example of two TTs requesting Reduce tasks, R_0 and R_1, from JT. TT_j stands for a Task Tracker node *j*. $IO_{Ri:TTj}$ stands for an intermediate output IO produced at TT_j and hashed to Reducer, R_i. JT might receive requests from TT0 and TT3 for Reduce tasks. Here are the two possible scenarios.

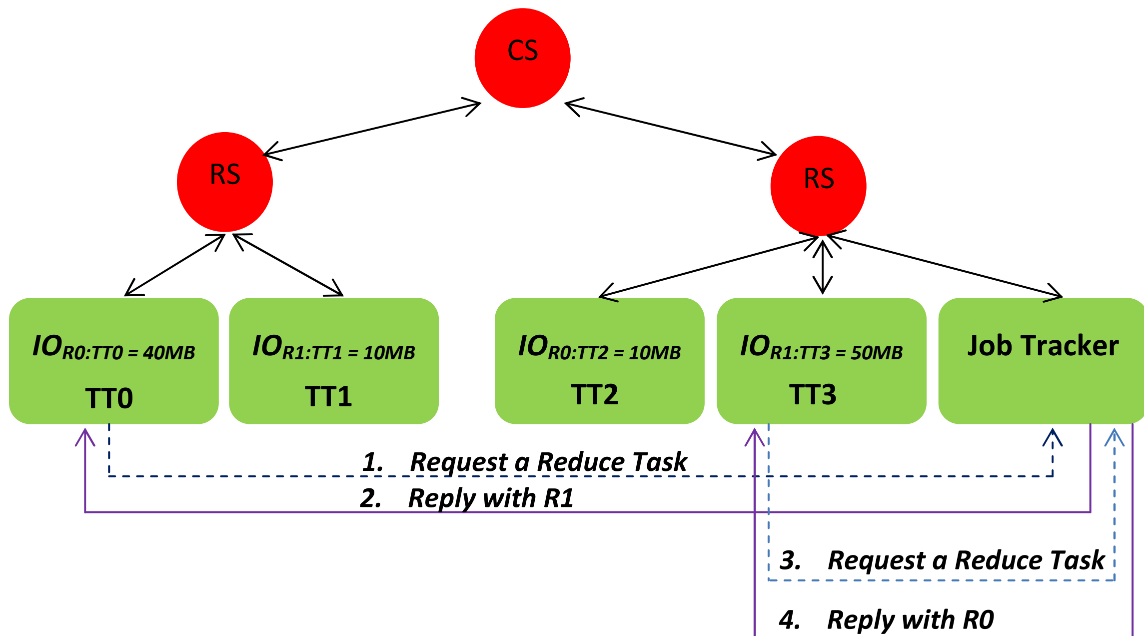

Figure 2: **An Example of Two Task Trackers making requests for Reduce tasks in native Hadoop (TT_j = Task Tracker j, R_i = Reduce i, and $IO_{Ri:TTj}$ = Intermediate Output produced at TT_j and hashed to reducer, R_i)**

Scenario I: JT replies to TT0 and TT3 with R1 and R0, respectively. This incurs shuffling $IO_{R1:TT1}$ and $IO_{R1:TT3}$ to TT0 from TT1 and TT3, respectively. In addition, $IO_{R0:TT0}$ and $IO_{R0:TT2}$ will be shuffled to TT3 from TT0 and TT2, respectively. As a result, 110MB will be shuffled on the cluster network.

Scenario II: JT replies to TT0 and TT3 with R0 and R1, respectively. This incurs shuffling $IO_{R0:TT2}$ from TT2 to TT0 and $IO_{R1:TT1}$ from TT1 to TT3. Consequently, only 20MB will be shuffled on the cluster network leading to 90MB decrease in network traffic as compared to Scenario I.

Hadoop, in its present design, is incapable of making controlled scheduling decisions similar to the one in Scenario II in order to diminish network traffic and improve performance.

VIII- CoRTS: A New Reduce Task Scheduler for Hadoop

As described in the previous section, the amount of data shuffled depends on where Reducers are scheduled. In case a reducer, R, has only one feeding mapper, M, the best data locality can be achieved via scheduling R at TT that hosts M. However, R typically has multiple feeding mappers which are usually located at multiple nodes. In this project, we suggest that good data locality can be achieved via scheduling R at the rack that holds the largest input size of R. Hence, the largest amount of data to be fed to each Reducer is always kept rack local; decreasing thereby the amount of data shuffled off-rack (off-rack traffic is incurred after a map output is shuffled to R from an off-rack feeding node). As Hadoop assumes a hierarchal network in which each rack contains a local switch and the rack switches are interconnected via a single core switch, communication between computers in the same rack becomes *cheaper* than communication between racks. Therefore, we suggest that such a strategy will provide Hadoop with better performance. We refer to this strategy as Controlled Reduce Task Scheduling (CoRTS) as it exerts some control on network traffic by avoiding shuffling large amounts of data across racks. Given the example shown in Fig. 2, if CoRTS is applied, Scenario II will be enforced. To this end, Algorithm 1 shows CoRTS.

Algorithm 1: CoRTS Algorithm

Input:

RT = set of unscheduled reduce tasks

TT = the task tracker requesting a reduce task

Output: A reduce task R Є RT that can be scheduled at TT

1. **Initialize a set of potential reducers to schedule at TT, $set_{offRack}$ = Φ**
2. **for every reduce task R Є RT do**
3. **LR = the rack that holds the largest input size for R**
4.
5. **if TT Є LR then**
6. **return R**
7. **else**
8. **add R to $set_{offRack}$**

9. **end for**
10.
11. **if $set_{offRack}$ is not empty then**
12. **return a random reducer R Є $set_{offRack}$**
13. **end if**

IX. Implementation Guidelines

In this project you will implement CoRTS in the Hadoop framework. The Hadoop framework includes hundreds of java classes. Hence, we suggest that you focus *mainly* on the following classes to implement CoRTS:

1) **JobTracker**: This is the central location class for submitting and tracking MapReduce jobs in a distributed environment. It represents the Master node in MapReduce.
2) **TaskTracker**: This is a process that starts and tracks MapReduce tasks in a distributed environment. It contacts the JobTracker for task scheduling and reporting results.
3) **JobInProgress**: This class maintains all the information for keeping a Job on, straight and narrow. It keeps its JobProfile and its latest JobStatus, plus a set of tables for doing bookkeeping of its tasks. This class contains the main code for the Hadoop task scheduler.
4) **TaskInProgress**: This class maintains all the information needed for a task in the lifetime of its own Job. A given TaskInProgress (TIP) contains multiple task ids, 0 or more of which might be executing at any one time (that is what allows speculative execution.) A TIP allocates enough task ids to account for all the speculation and failures it will ever have to handle. Once those are up, the TIP is dead.
5) **MapTask**: This class represents a map task in MapReduce.
6) **ReduceTask**: This class represents a reduce task in MapReduce.
7) **Task**: This class is the superclass of classes MapTask and ReduceTask.
8) **TaskStatus**: This class describes the current status of a task.

In order to simply your job, we have already modified some of the classes and coded parts of the algorithm for you. Specifically,

1) We modified the **Task** and **MapTask** classes to keep track of reducers per mapper (i.e., all reducers that a mapper feeds) as well as the intermediate output size produced by every mapper to every consuming reducer.
2) We modified the **TaskStatus** class to reflect the new changes on the status of the MapTask (as new variables and data structures have been added). This way, when a status of a map task is reported to the JobTrakcer (via the heartbeat protocol), the new collected information is sent. The new collected information (i.e., reducers per mapper and input sizes to reducers from feeding mappers) will be needed by CoRTS at JobTracker in order to figure out the rack that holds the largest intermediate output for a reducer.
3) We also modified the **TaskTracker** class to update the data structures that keep track of reducers per mapper and input sizes to reducers.
4) We modified the **JobTracker** class to receive (through the heartbeat mechanism) the new collected information at the **TaskTracker** class.
5) We modified the **JobInProgress** class to make the new collected information available to the Hadoop scheduler. In particular, we made the following modifications to this class:

    a. We implemented a function required by CoRTS that computes the mappers per each reducer using the information collected at the JobTracker class (i.e., the reducers per each mapper).
    b. We implemented a function that returns the racks (represented by their integer numbers) that hold the feeding nodes of a given reducer. You should utilize this function to implement another function that returns the rack that holds the largest intermediate outputs for a given reducer.
    c. We also made many other modifications including, but not limited to, functions that allow you to print statistics about the collected and the computed data.

Please note that all modifications that have been made to the Hadoop framework are tagged by MHH_Code. Hence, you can easily read all added and modified code by searching for MHH_Code.

Now, we assume the following:

- A Hadoop cluster with two racks, each with two nodes.

- The nodes in the cluster have large disks directly attached to them, allowing application data to be stored on the same computers on which it will be processed.

As such, we also provide you with the following:

- A cluster of 4VMs, each with 1vCPU, 1GB RAM, 20GB storage and Fedora 15 64-Bit operating system. Two of your VMs go under one rack and the other two under another rack.
- Hadoop 0.20.2 already set up and ready for you with the modified classes being integrated and network topology being configured.

X. Experimentation and Analysis

Use the sort benchmark available in the Hadoop examples jar file to conduct your experimental studies. Please provide the following:

1) A comparison between Hadoop with its native scheduler and Hadoop with CoRTS in terms of:
    a. Total execution time.
    b. Total amount of data shuffled.
    c. A breakdown of data (i.e., whether local, on-rack or off-rack data) consumed by reducers. We have coded this function for you. You can collect this data simply by reading the file *your_Job_ID_ _shuffleStat_Results.txt* present under your logs directory.
2) A discussion on:
    a. The collected results.
    b. Your experience in modifying the Hadoop framework.

c. Your insights concerning distributed scheduling in general and Hadoop task scheduling in particular.
d. How CoRTS scales with deeper network hierarchy. That is, do you think CoRTS would extend trivially to networks with more racks and nodes per racks? If not, can you suggest an extension to CoRTS (or a completely different strategy) so that it can supposedly scale better? (Please note that this is merely a conceptual discussion. We do not require you to conduct any scalability study for this project)

3) **BONUS (25 Points):** An Extension to CoRTS

a. Propose and implement an extension to CoRTS that could account for more locality and, consequently, better performance.
b. Compare your proposal with native Hadoop and CoRTS in terms of:
    i. Total execution time.
    ii. Total amount of data shuffled.
    iii. A breakdown of data.
c. A discussion on your collected results.

XI. Deliverables

Final Deliverables

As final deliverables, you should submit:

- An archive containing all your code (only the Hadoop classes that you modified).
- A minimum of 3-page article (similar to research articles) that presents your solution, findings, observations and analysis.

On December 13, we will hold a presentation session (during the normal class time) where each team will present (for 25 minute using PowerPoint slides) its work to the class.

XII. Handing In the Project

Submit your documents and code on AFS directory:

/afs/qatar.cmu.edu/msakr/www/15440-f11/handin/userid/p4/,

where ‘userid’ is your andrew user ID.

XIII. Late Policy

- If you hand in on time, there is no penalty (duh!).
- 0-24 hours late = 25% penalty.
- 24-48 hours late = 50% penalty.
- More than 48 hours late = you lose all the points for this project.

NOTE: You CANNOT use your grace-days quota for project 4. For details about grace-days quota, please read the syllabus.

XIV. Team Project Policy

This project is a team project. The class will be split into teams of two students. Each student can select her/his teammate. Each team should divide the work, including coding, testing, analysis, write-up, and presentation, equally among all its members. Members of a team might get different grades on the project if after evaluation we recognize that the efforts put by members clearly vary. A presentation session will be held where each team will present for 25 minute its work to the class. After submitting the project, each team has to schedule an appointment with one of the course instructors in order to discuss the delivered work.