An Investigation of Efficient Task Assignment in Hadoop

Girish Sastry

February 3, 2011

1 Introduction

1.1 Motivation

Until recently, most computing was done on a single computer, with its memory, local disk, and cache. However, with the rise of Internet platforms and web applications, data on the order of petabytes needed to be stored and analyzed. This data is too large to fit on a single computer, and sequential algorithms to analyze the data would be extremely inefficient. Google’s MapReduce is a large scale data processing architecture that seeks to solve the problems of scalability and efficient computation when dealing with large amounts of data. Hadoop is the open source implementation of MapReduce, and is an important part of the software stack at many companies, such as Facebook, Yahoo!, Twitter, and Amazon.

1.2 Architecture

Hadoop (and MapReduce) take advantage of networked clusters of commodity computers to store and analyze data. In order to store data across nodes, a distributed file system (Hadoop File System, or HDFS) partitions the data among the cluster of nodes. Physically, these compute nodes are stored on racks, with each rack containing some number of networked nodes (typically up to 64). Many racks communicating with each other can form a cluster. Communication between racks has a larger latency than intrarack communication.

It is inevitable that hardware fails and that computations may need to be rerun. This could be a very costly operation, and Hadoop tackles this problem in two ways: first, by assuring that files are stored redundantly (fault tolerance), and second, by splitting each computation up into atomic “tasks”. If a particular task fails, it can be restarted without affecting any of the other tasks.

Computations follow the Map-Reduce paradigm, which allows for easy parallelization of computations. A map-reduce job is specified by tasks. A user needs only to write two functions: Map and Reduce, and the system handles the parallelization and execution. Map tasks read chunks of input from HDFS and operate on them to produce a list of intermediate key-value pairs. A sort step then occurs in which a master controller node sorts the list of intermediate key-value pairs by key. Then, the reduce tasks operate on one key at a time, performing the user defined reduce function on all values associated with that
particular key. Finally, the output is written in the same manner as the input was stored.

1.3 The Assignment Problem

The Hadoop scheduler assigns map tasks. Assignment of map tasks is a crucial process that affects the efficiency of the overall system. Each reduce task cannot begin until it receives the intermediate list of key-value pairs from a map task. Furthermore, the assignment of the map task sets the location of the storage of the intermediate data. Thus, optimal assignment of map tasks is a goal of Hadoop’s scheduler.

Su et al. have shown the problem of optimal task assignment in Hadoop is NP-complete. They have also shown some interesting theoretical results about the default Hadoop scheduler’s round robin algorithm: that increasing the number of data block replicas may increase the maximum load computed by the round robin algorithm. Finally, Su et al. propose a network flow based algorithm that runs in time $O(m^2n)$, where $m$ is the number of tasks and $n$ is the number of servers.

2 Experimental Goals

The theoretical model for Hadoop proposed by Su et al. is elegant and concise and provided important insights for the development of their flow based scheduling algorithm. However, real world systems deviate from models for a variety of reasons. We propose to perform experiments with a few goals: first, to test the validity of the assumptions made in the theoretical model, second, to optimize the proposed flow based algorithm, and third, to analyze behavior of the Flow-algorithm with real world data and use cases.

2.1 Optimizing Flow-Scheduling

The Flow-scheduler algorithm works in two parts: first by running a Ford-Fulkerson network flow algorithm to compute a partial assignment of tasks to servers, and then by greedily assigning the rest of the tasks. Generally, we would like to run the network flow algorithm when tasks are accessing local data (cost of access is low, $w_{\text{local}}$), and greedily assign the tasks that have to deal with remote access ($w_{\text{remote}}$). This is because the network flow algorithm is the main culprit in terms of time efficiency, and we would like to use this with tasks that have the least access latency. However, Su et al. assume that the cost of access for local disk is equal to the cost of access for same rack servers. In reality, this is not the case. Thus, we hope to experimentally answer the question: should we use network flow to assign same rack tasks or should we greedily assign them?

There are other potential optimizations to the Flow-scheduler algorithm that we hope to investigate. For example:

- How does $\frac{w_{\text{remote}}}{w_{\text{local}}}$ change given a varying network and traffic pattern?
- Online vs Offline optimization: what happens when we don’t know all the tasks or available servers ahead of time?
• What are good input data placements (where to store the data?) that help us assign tasks efficiently?
• How can we achieve these placements and can we maintain them during execution of the algorithm?

2.2 Theoretical bounds

We would also like to verify the worst case bound proved by Su et al. Performing this real world analysis would also serve reinforce some of the assumptions made to formalize the Hadoop task assignment problem. Do we see the worst case analysis approach the bound proposed by Su et al.? Since the Flow-scheduling algorithm is an approximation algorithm that is optimal within an additive constant, we would like to investigate how this constant varies under real world data. Furthermore, there is no average case analysis in the theoretical work of the Flow-scheduler algorithm because there were too many parameters and a theoretical average case analysis didn’t make sense. We hope to shed some light on the “average” case behavior of the Flow-scheduler algorithm (and also define what “average” even means in the context of Hadoop).

3 Deliverables

We will have a report detailing the experimental procedure, goals, hypotheses, etc. We will also make available all data, experimental configurations, data analysis, and code used in the project.

References


