Master’s Thesis: A Tuning Approach Based on Evolutionary Algorithm and Data Sampling for Boosting Performance of MapReduce Programs

Tiago R. Kepe¹, Eduardo Cunha de Almeida¹

¹Programa de Pós-Graduação em Informática (PPGINF)
Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brazil
{trkepe,eduardo}@inf.ufpr.br

- Master’s admission year: 07/2012
- Proposal’s defense date: 11/2013
- Expected finish date: 12/2014
- Concluded steps:
  - Implementation;
  - Initial results.
- Future steps:
  - Generating relevant results;
  - Analysing the new results;
  - Writing the thesis.

Abstract. The Apache Hadoop data processing software is immersed in a complex environment composed of huge machine clusters, large data sets, and several processing jobs. Managing a Hadoop environment is time consuming, toilsome and requires expert users. Thus, lack of knowledge may entail miscalculations degrading the cluster performance. Indeed, users spend a lot of time tuning the system instead of focusing on data analysis. To address misconfiguration issues we propose a solution implemented on top of Hadoop. The goal is presenting a self-tuning mechanism for Hadoop jobs on Big Data environments. To achieve this, our self-tuning mechanism is inspired by two key ideas: (1) an evolutionary algorithm to generate and test new job configurations, and (2) data sampling to reduce the cost of the self-tuning process. From these ideas we created a framework for testing usual job configurations and get a new configuration suitable to the current state of the environment. Experimental results show gains in job performance against the Hadoop’s default configuration and the rules of thumb. Besides, the experiments prove the accuracy of our solution which is the relation between the cost to obtain a better configuration and the quality of the configuration reached.

Keywords: Big Data, MapReduce, Hadoop, Self-Tuning
1. Introduction

Nowadays, companies and research scientific institutes are awash in a flood of data where "Big Data" has emerged. A standard solution and vastly used for Big Data purposes is the programming framework MapReduce (MR). It was an initiative of the Google company in 2004 when they disclosed the MapReduce framework [Dean and Ghemawat 2004] for storing and processing data on large clusters.

A popular open source implementation of the MR framework is the Apache Hadoop which has an intuitive interface to create MR jobs. Hadoop orchestrates parallel processing, handles node and task failures, stores data in a distributed way, recovers data and so on. In addition, users ought to properly setup a MR job because misconfiguration may degrade the job performance [Kambatla et al. 2009]. However, the Hadoop’s environment is complex and there is a huge number of parameters to be configured for each job, such as: settings for memory allocation, I/O controllers, network timeouts etc. Performance and resource usage of jobs are directly bound to these configuration parameters. Normally, a job is configured manually by developers or testers, which is obviously time consuming as the process has to be iterative: configuration of job, execution of the job, measurement of performance, re-configuration of the job and so on. Accomplishing tuning on real Hadoop environments becomes impracticable due to the large amount of jobs, the high number of parameters involved larger than 250 and the cluster "dynamism”.

2. Contribution

To perform self-tuning of MR jobs we were inspired by two key ideas: the Bacteriological Algorithm (BA) [Baudry et al. 2005] using the job execution time as fitness value, and data sampling which led us to create a dynamic reservoir sampling algorithm called KSample. Whilst the BA generates new job configurations along the execution, KSample reduces the cost to test these configurations.

2.1. Algorithm for testing

The Bacteriological Algorithm (BA) belongs to the family of evolutionary algorithms. We adapted the BA to work in the genetic context. Our BA implementation has the gene as atomic unit, a group of genes forms an individual (bacterium), and a group of individuals forms a population (bacteria). As Hadoop is involved in a peculiar environment, we performed a context transformation to adapt the BA to work on the Hadoop context. We called a specific configuration parameter as knob, an entire setup as set of knobs and the combination of setups as sets of knobs. In the context transformation, each component of the genetic context was translated to a component of the Hadoop context. A gene was transformed to a knob, an individual was transformed to a set of knobs, and individuals population was transformed into sets of knobs.

The BA in high-level of abstraction is described in the Algorithm 1. It receives as input an initial sets of knobs, the number of generations to be reached and the desired fitness value. (1) The following process computes the fitness value for each set of knobs: initially, a job is configured with the current set of knobs and sent to be executed over a data sample. Then the job execution time is assigned as the fitness value for this set of knobs. (2) The best set of knobs reached is memorized based on the fitness value. (3) The reproduction operator creates a new generation composed of clones of the best set of
knobs. (4) The mutation operator iterates on the new population, and for each set of knobs one or more knobs are randomly selected and their values are changed randomly. The process continues until the number of generations or the desired fitness value is reached.

<table>
<thead>
<tr>
<th>Algorithm 1: Bacteriological Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: Pop Initial population</td>
</tr>
<tr>
<td><strong>Input</strong>: Gen Number of generations</td>
</tr>
<tr>
<td><strong>Input</strong>: Fitness Desired fitness value</td>
</tr>
<tr>
<td><strong>Output</strong>: BestKnobs The best set of knobs reached</td>
</tr>
<tr>
<td>repeat</td>
</tr>
<tr>
<td>for each knobs ∈ Pop do</td>
</tr>
<tr>
<td>CalcFitness(knobs)</td>
</tr>
<tr>
<td>BestKnobs ← Memorization(Pop, BestKnobs)</td>
</tr>
<tr>
<td>Pop ← Reproduction(BestKnobs)</td>
</tr>
<tr>
<td>Pop ← Mutation(Pop)</td>
</tr>
<tr>
<td>until Gen ∨ Fitness</td>
</tr>
<tr>
<td>return BestIndiv</td>
</tr>
</tbody>
</table>

### 2.2. Sampling on MR

The BA’s process allows to create new configurations and test them in order to select which one is more suitable to the MR job at the system’s current state. However, running the algorithm in the whole data set may be unfeasible, especially for big data. Thus, to achieve automation of testing, it is crucial to apply techniques that efficiently reduce the amount of data to be processed. A common approach used on relational DBMSs is data sampling. However, on Big Data context, data sampling is challenging due the large amount of data, not only the sampling must be done in a distributed fashion, but also the storage.

Thus, according to [Vitter 1985, Cloudera, Inc 2013, Grothaus 2013] unstructured data stream, which is common for MR applications, can be addressed by Reservoir Sampling. However, the size of the reservoir can be challenging to find due to the size of data. Then, we created a new algorithm called (KS)Sample which is inspired by the Reservoir Sampling algorithm. KSample receives a percentage (as parameter) of the population to be sampled. For instance, 10% is received as input, then KSample will create a reservoir sample that represents 10% of the input data records. The KSample works on unstructured data, the atomic unit for the algorithm freely can be defined such as: record, row, byte, file, etc. The KSample works with an undefined reservoir size and increases the reservoir sample on demand.

We proved mathematically, see Appendix A, that the KSample algorithm holds at least $\rho\%$ of the input data. We built the KSample as a MR application in order to execute distributively. Basically, the map function receives the key/value pair $\langle lineNumber, lineContent \rangle$ and generates the intermediate pair $\langle randomNumber, lineContent \rangle$. MR sorts the intermediate pairs and aggregates the values that share the same key, then the reduce receives the pair $\langle key, list<lineContent> \rangle$ and runs the KSample locally.
Algorithm 2: KSample Algorithm

**Input**: percentage percentage for sampling  
**Input**: stream data stream with undefined length  
**Output**: reservoir

\[
\text{sLength} \leftarrow 0 \\
\text{slotRound} \leftarrow 0 \\
\text{while} \ stream \neq \text{EOF} \text{ do} \\
\quad \text{sLength}++ \\
\quad \text{if} \ reservoir.\text{size}() < (\text{percentage} \times \text{sLength}) \text{ then} \\
\quad \quad \text{reservoir.}\text{newSlot}() \\
\quad \quad \text{slotRound} \leftarrow 0 \\
\quad \text{slotRound}++ \\
\quad \text{probability} \leftarrow \frac{1}{\text{slotRound}} \\
\quad \text{rand} \leftarrow \text{Random}(0, 1) \\
\quad \text{if} \ \text{rand} \leq \text{probability} \text{ then} \\
\quad \quad \text{reservoir.}\text{currentSlot} \leftarrow \text{stream}[\text{sLength}] \\
\text{return reservoir}
\]

2.3. Framework for testing

We created a framework for testing the configuration setup composed of three main modules: the front-end for users interacting with the system using a Domain-Specific Language (DSL), the engine to test job configurations and the back-end that saves the best configuration found by the engine.

The engine is composed of three sub-modules:

- Sampler: generates data sampling by invoking the KSample;
- Core: generates job configurations using the BA;
- AutoConf: enforces Hadoop applying the configurations generated by the Core.

An overview of the framework is presented in the Figure 1. First, (0) an user creates a file following our DSL, (1) this file is submitted to the Front-end. (2) The Analyser is invoked to validate the DSL code. (3) The parsed file is sent to the Engine component, it actives the Sampler module (4). After the core starts the BA process that will generate new configurations and send them to the AutoConf (5). The Core submits the job to the Hadoop (6), when Hadoop queues the job for execution, it sends a request to the AutoConf, which returns the current configuration for the job. The steps 5, 6 and 7 occur until the BA process is finished. In the end the resulting configuration is sent to the Back-end. So, users will have a new configuration suitable for the current state of the environment or at least a good trade-off about the job settings.

3. Related tuning techniques for MapReduce

Some rules of thumbs have been created to adjust Hadoop environments based on administrators and developers’ knowledge [White 2009, Cloudera 2014, Intel 2010]. Applying the rules is straightforward, but not individually accurate. The rules are generic, aiming to be applied in all jobs without regard to specific job behaviors.
Another solution is profile-driven tuning which is based on job profiles using performance models, such as Starfish [Herodotou et al. 2011, Herodotou and Babu 2011]. Profile-driven tuning requires a previous job execution, when a new job is submitted, Starfish runs it and uses dynamic instrumentation to collect statistical data. The drawback is performance overload caused by this instrumentation.

Another alternative is to use simulators such as MRPerf [Wang et al. 2009], WaxElephant [Ren et al. 2012] and SimMapReduce [Teng et al. 2011]. The simulators emulate real big data environments, but they require extra information such as: existing workloads, log files and other relevant sources. However, they might not simulate events that only happen in the real world, such as: cluster volatility, i.e. nodes entering and leaving of the cluster, data dynamism that can be delivered in new formats, etc.

4. Related data sampling approaches for MapReduce

BlinkDB is a query engine for running interactive SQL queries on large volumes of data. It focuses on running short exploratory queries to provide trade-off query accuracy for response time [Agarwal et al. 2013]. BlinkDB creates a set of stratified samples based on the past queries behaviors. EARL [Laptev et al. 2012] is another framework for Hadoop to run queries on samples. It uses the bootstrapping sample technique, but the sampling algorithm requires some assumptions set by the users, such as: the size and the number of samples.

Those data sampling approaches requires prior knowledge about data or the query history. However, we are interested in using a naive input for sampling in order to abstract the complexity from the users, like a percentage of the data population.
5. Initial Results

We performed initial experiments on Hadoop with 1 master machine and 2 slave machines. These 3 machines ran Linux SO with 3GB of memory. Each slave machine was populated with 10GB of random texts using the job `randomtextwriter` embedded in the Hadoop example jar file. We ran 10 generations (rounds) of the BA using the WordCount job on a sample of 20% percent of the data (4GB). We saved the best configuration reached in each round and applied them in the whole data set (20GB) to measure the quality of the configurations. We also ran the BA without sample to compare with our tuning approach.

Figure 2 shows the quality of our approach against the default Hadoop configuration and the rules of thumb applied on the WordCount job. The configuration reached in the round #10 had a gain in performance of 10% compared with the default configuration and almost 25% against the rules of thumb. The same graphic shows that the best configuration reached with and without sampling had a similar performance, but the cost using data sampling is significantly lower, as can see in Figure 3, where the BA without sample was 6 up to 7 times less costly.

6. Future work

For future work we plan to perform more experiments on a large cluster to corroborate the accuracy of our tuning approach. We have considered just the job execution time as fitness value, we should be considering other metrics like: IO operation, CPU, Memory and Network usage. In cloud computing premises, users might be more concerned about resource usage than in response time, as cloud service providers offer plans based on "pay as you go", such as Amazon EC2. We are also studying to implement other algorithms to perform MR self-tuning, such as: hill climbing and simulated annealing.

References

A. KSample’s proof

KSample is based on the fact that the reservoir always holds at least a gave percentage of elements. That is the invariant property of the KSample, i.e. independent of the current stream length, it ensures that in any step the reservoir always will hold at least a gave percentage of elements that arrived. Also, the KSample ensures that every element has the same probability to be inserted into the reservoir which is inherited from the reservoir algorithm, as long as it treats any slot as a mini-reservoir of size 1. The invariant property is ensured by proofing that KSample creates a new slot at right moment, i.e. if the KSample didn’t create a new slot, the reservoir wouldn’t contain at least the percentage of elements from the stream. Let’s prove that property by induction:
• Notations:
  R: reservoir size.
  P: percentage of the stream.
  L: stream length.

• Base Case: When \( E_1 \) (first element) comes.
  \( R = 0 \), the algorithm has to decide in creating or not a new slot, for this it checks the condition \( R < (P \times L) \).
  As \( P \in [0,1] \) and \( L = 1 \), then \( (P \times L) \in [0,1] \).
  Consequently, \( R < (P \times L) \) is true and a new slot is created, thus the reservoir will hold \( E_1 \) and it will have at least \( P \) percentage of elements from the stream.

• Induction Hypothesis (I.H.):
  Suppose in the step \#N after the \( E_n \) element arriving at the reservoir holds at least \( P \) percentage of elements from the stream.

• Induction Step: step \#(N + 1).
  We have to prove two cases:
  1. Create a new slot:
     For this case the condition: \( R < (P \times L) \) has to be true. By the I.H. in the last step (step \#N) the reservoir holds \( P \% \) of elements from the stream, as the KSample will create a new slot and will hold the element \( E_{n+1} \) (following the reservoir sample algorithm), then, certainly, adding a new slot the reservoir size will increase, thus it will continue holding at least \( P \% \) of elements from the stream.
  2. Don’t create a new slot:
     The KSample decided not to create a new slot, then \( R \geq (P \times L) \), means that the reservoir is holding \( P \% \) or more elements from the stream, otherwise would be the case 1.

Therefore, after the step \#(N + 1) the reservoir size is invariant in any round of the KSample, because it will contain at least \( P \% \) of elements from the stream.