Execution Time Prediction of Jobs Using SWAT

Omar Arafa
Alexandria University
omarkhaledarafa@gmai.com

Hisham Raslan
Alexandria University
eng.hisham.raslan@gmail.com

Prof. Iman Elghandour
Alexandria University
ielghand@alexu.edu.eg

ABSTRACT

The current high workload and lengthy jobs deployed on the cloud infrastructure needs to be configured with the appropriate hardware configurations in order to give the best performance for these workload. Many different hardware configurations are available in cloud computing infrastructure. Subtle changes in these configurations could lead to a significant gain in the performance of different workloads. Hardware configurations like number of cores, number of machines, memory size and number of GPUs provide a powerful yet hard to configure environment. The combination of lengthy jobs and different hardware configurations lead to the need of efficient predictive models for jobs execution time. GPUs are powerful in doing parallel computations and many distributed systems framework are providing the facility of using them during their computations. SWAT is a framework for running spark jobs on GPUs. we used SWAT as it provide minimal code change of spark job to run it on GPU. We introduce a predictive model that predict execution time mainly using SWAT but it can be generalized on any spark job. We achieved a decent accuracy using the gradient boosting regressor around 76%.

CCS CONCEPTS

•Distributed Systems → Big Data; •Machine Learning → *Regression;* **•Parallel Programming** → GPGPU;

ACM Reference format:
Omar Arafa, Hisham Raslan, and Prof. Iman Elghandour. 1997. Execution Time Prediction of Jobs Using SWAT. In *Proceedings of ACM Woodstock conference, El Paso, Texas USA, July 1997 (WOODSTOCK'97),* 5 pages.
DOI: 10.475/123_4

1 INTRODUCTION

In recent years, there has been huge deployment of large scale advanced analytics that implements compute-intensive algorithms in various areas like deep learning [1], genome analysis [3] and traditional data intensive workloads like SQL queries. Most of these workloads typically takes long time to process and hard to configure hardware in order to achieve optimal execution. Hence, the emergence of predictive models for such workload on different hardware configurations in order to decide which provide the best performance. Currently, there are many hardware configurations available for using in cloud infrastructure that varies in number of machines used, number of cores in each machine, the size of memory, the size and type of persistent storage(for example: SSD) and recently the number and type of GPU used. Lately, there has been a significant increase in usage of GPGPU(general purpose gpu) in cloud infrastructure as they are more computationally than traditional CPUs. Many frameworks has begun to exploit the computation power of GPU in distributed system like tensorflow, SWAT, HeteroDoop, SparkCL and HadoopCL. These frameworks usually runs lengthy jobs that has various hardware configurations available for it. So, the need of accurate predictive models for these framework is a must. Many paper recently tried to tackle these problems in different ways. Some papers tried to treat the job as a black box and by knowing the common workload execution parameters and behaviors it try to predict the execution time on whole set of data using the execution times of samples from the data, This approach was adapted by Earnest [4]. Other approaches predict the GPU execution time using the execution time of CPU. these approaches has been adopted in [5] which uses a machine learning algorithms in the prediction of GPU execution time. Also, Statistics driven workload prediction has been popular since the emergence of [2]. The paper will proceeds as follows in section 2, we will demonstrate necessary background in order to achieve more inference in our approach. In section 3, we discuss our predictive model and its various aspects. In section 4, we provide different experiments and evaluation of our model. Finally, in section 5, we conclude our work and illustrate aspects of future work.

WOODSTOCK'97, El Paso, Texas USA

DOI: 10.475/123_4

2 BACKGROUND

2.1 GPUs

GPUs are a very powerful in processing parallel computations. For long time, GPUs were only used in performing graphics computations only. The emergence of GPGPU (general purpose gpus) has opened the door for usage of GPUs in compute-intensive problems. It should be apparent that Spark is a highly parallel framework for processing large amounts of data. This computational pattern fits well with GPUs. Unfortunately, Working with GPUs require low-level knowledge about the architecture of GPU itself, its memory structure and parallel model in order to implement highly efficient programs. GPU memory hierarchies also contain special-purpose memory that allow programmers to manually place data with certain characteristics in appropriate parts of the hierarchy. For example, GPU constant memory is useful for constant data structures read by all threads on the GPU. GPU shared (or scratchpad) memory is on-chip and useful for storing frequently accessed, small data structures. Different GPU models are available in order to provide maximum hardware performance, the most commonly used GPU models are CUDA and OpenCL. In contrast to OpenCL, CUDA only works on Nvidia GPUs while OpenCL works on different kind of GPUs and accelerators such as FPGAs and DSPs. Yet, these models require intensive knowledge of parallel programming. The data

parallelism, memory bandwidth, and low-level programming models of GPUs make them complementary to data analytics platforms like Spark. SWAT provides the facility to exploit GPUs in spark framework without the need of intensive knowledge in parallel programming.

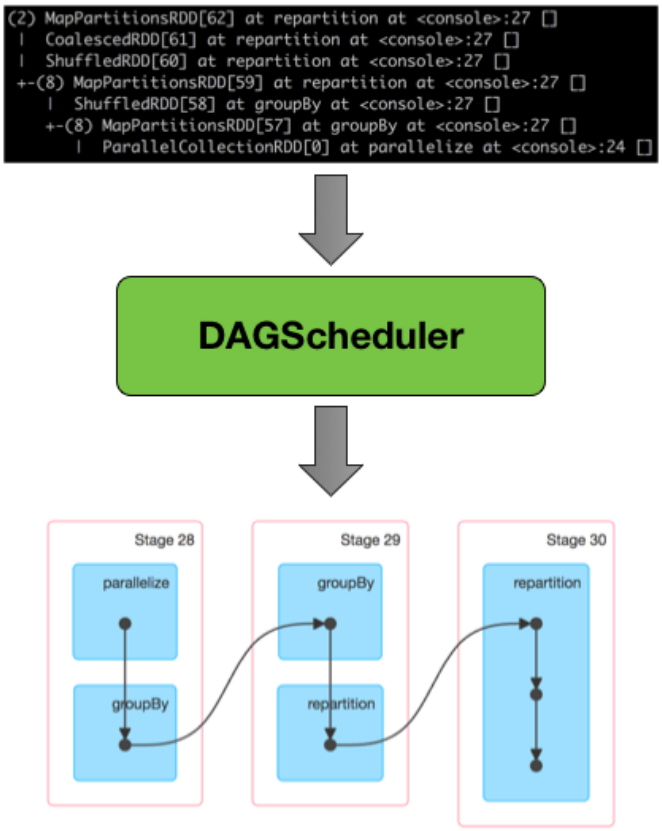
2.2 Spark

Apache Spark is a distributed, multi-threaded, in-memory programming system. The core abstraction of Apache Spark is that of a resilient distributed dataset (RDD). An RDD represents a distributed vector of elements. Elements in an RDD can be of any serializable type. RDD creation is lazy: creating an RDD object in a Spark program does not necessarily evaluate and populate that RDD. Only certain operations in Spark programs force evaluation, resulting in long chains of lazily evaluated RDDs as one RDD is transformed into another. RDD resiliency derives from their ancestry tracking. By persisting information on how RDDs were created rather than their actual contents, Spark guarantees that lost data can be recovered through recomputation without storing large amounts of intermediate data on disk. A single RDD is split into multiple partitions. All elements in the same partition are stored on the same machine, but different partitions may be stored on different machines. Hence, partitions are the granularity of distribution in Spark. One of Sparkfis strengths is its API, i.e. the transformations that it supports on RDDs. Spark transformations run in parallel across the machines that an RDD is stored on. Transformations are functional: they are applied to one RDD and produce another. This generally leads to long chains of lazily evaluated RDDs, linked by functional transformations. The transformations that Spark supports include map, reduce, filter, reduceByKey, groupByKey, join, and distinct. This variety of transformations greatly expands the flexibility of Spark relative to its predecessor, Hadoop MapReduce. DAGScheduler is the scheduling layer of Apache Spark that implements stage-oriented scheduling. It transforms a logical execution plan (i.e. RDD lineage of dependencies built using RDD transformations) to a physical execution plan (using stages). A stage is a physical unit of execution. It is a step in a physical execution plan. A stage is a set of parallel tasks, one per partition of an RDD, that compute partial results of a function executed as part of a Spark job. DAGScheduler splits up a job into a collection of stages. Each stage contains a sequence of narrow transformations that can be completed without shuffling the entire data set, separated at shuffle boundaries, i.e. where shuffle occurs. Stages are thus a result of breaking the RDD graph at shuffle boundaries. Figure 1 shows a high level diagram of how the DAGScheduler operates.

2.3 SWAT

SWAT is a programmable, in-memory, distributed, high-performance computing platform. SWAT is an accelerated data analytics (ADA) framework that enables programmers to natively execute Spark applications on high performance hardware platforms with co-processors, while continuing to write their applications in a JVM-based language like Java or Scala. Runtime code generation creates OpenCL kernels from JVM bytecode, which are then executed on OpenCL accelerators. In our work we emphasize 1) full compatibility with a modern, existing, and accepted data analytics platform,

Figure 1: DAGScheduler Transforming RDD Lineage Into Stage DAG.



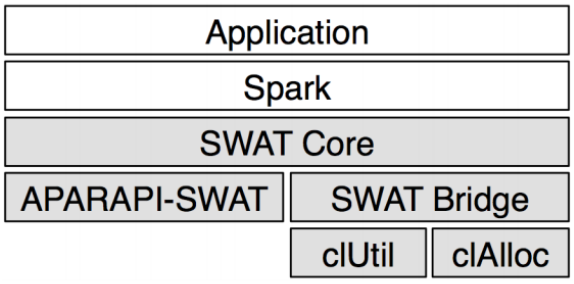
2) an asynchronous, event-driven, and resource-aware runtime, 3) multi-GPU memory management and caching, and 4) easeof-use and programmability. SWAT is high-level: it does not expose accelerator hardware details, maintaining the same APIs as Spark. It fully integrates with the Spark framework as a third-party JAR, requiring only minimal code re-write for existing Spark applications. The open-source Spark With Accelerated Tasks (SWAT) framework, available at https:// github.com/agrippa/spark-swat. SWAT accelerates the user-written computational kernels of Apache Spark jobs using OpenCL-supporting GPUs. SWAT is not fixed function: it allows Spark programmers to write custom kernels in high level JVM languages like Scala. SWAT uses an asynchronous, event-driven, and resource-aware runtime to accurately and efficiently manage intra-node resources for the programmer. Figure 2 shows the SWAT software stack.

3 MODEL ARCHITECTURE

3.1 Overview

Our machine learning predictive model has two main phases training phase and test phase. During training phase, we train our machine learning model on features extracted from jobs or stages,

Figure 2: The SWAT software stack.



data and hardware configurations. Then in testing, we extract these features and predict the execution time of the job or stage itself. Currently, two types of models has been examined during the project. First, the full predictive model which is trained on the feature vector extracted from the job or stage after executing on the whole data. The other model is trained to extrapolate the execution time by running a small sample of data which we refer to it subsequently as the extraplator model. The features of each job or stage is extracted from the spark execution logs. We currently provide two predictive models: one of them used to predict stage execution time and the other is used to predict the whole job execution time.

3.2 Job and Stages Features

We used a set of features in order to identify each job uniquely from the others. In stage predictive model, we extracted the features shown in table 1 from the spark's execution logs which is showed using spark's webui. Each stage is divided into set of tasks where these task aggregate data represents our feature vector, for example we count the number of RDD in all task, min, median and max of duration of each task, min, median and max of scheduler delays of each task, etc. The feature vector of the job is the aggregate feature vector of each stage adding to it the number of stages. These features are used to identify the whole state of a job or stage. Mostly, they could be splitted into two categories: Job and data features which is represented by number of RDDs, read and write data size, various types of execution times and hardware configurations represented in the number of cores, number of machines, number of GPUs and Memory size. These features are framework specific, they work only on swat-spark framework. A more general feature from the job could be obtained by analyzing the job code itself for example by analyzing java byte code of the job. Also, A thorough data analysis may be conducted in order to get better features for the data, for example constructing histogram for the data.

Feature	Description
Number of RDD	the total number of RDD in each stage or job.
GC Time	Time that the executor spent paused for Java garbage collection while the task was running.
Getting Result Time	Time that the driver spends fetching task results from workers. If this is large, consider decreasing the amount of data returned from each task.
Peak Execution Memory	Execution memory refers to the memory used by internal data structures created during shuffles, aggregations and joins when Tungsten is enabled. The value of this accumulator should be approximately the sum of the peak sizes across all such data structures created in this task. For SQL jobs, this only tracks all unsafe operators, broadcast joins, and external sort.
Result Serialization Time	Time spent serializing the task result on the executor before sending it back to the driver.
Scheduler Delay	Scheduler delay includes time to ship the task from the scheduler to the executor, and time to send the task result from the executor to the scheduler. If scheduler delay is large, consider decreasing the size of tasks or decreasing the size of task results.
Shuffle Read Blocked Time	Time that the task spent blocked waiting for shuffle data to be read from remote machines.
Shuffle Read Size / Records	Total shuffle bytes and records read (includes both data read locally and data read from remote executors).
Shuffle Remote Reads	Total shuffle bytes read from remote executors. This is a subset of the shuffle read bytes; the remaining shuffle data is read locally.
Task Deserialization Time	Time spent deserializing the task closure on the executor, including the time to read the broadcasted task.
Number of Stages	The total number of stages in a certain job only used in job predictive models
Number of machine	Number of machine used to the job.
Memory Size	Memory size used aggregated from all machines.
Number of GPUs	The total number of GPUs used in all machines.
Number of cores	the total number of cores used in all machines.

Table 1: Features used to identify certain stage or job used in training and testing our predictive models.

3.3 Machine learning Models

We used the same machine learning algorithms for job and stage execution time prediction. In general, any regression machine learning algorithm will be suitable for our execution time prediction. Yet, small data size limited us to certain simple algorithms in order to be able to generalize the model on test data easily and prevent overfitting.

3.3.1 SVM Regressor. We made an SVM regression model using job and stages datasets. SVM regression model formally known as SVR which perform non-linear regression by mapping the input into higher dimensional space using a non-linear kernel. We used guassian radial basis function as our kernel which is expressed by the equation $k(x_i, x_j) = \exp(\frac{-|x_i - x_j|}{2*\sigma^2})$.

3.3.2 Gradient Boosting Regression. Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

4 EVALUATION

All experiment has been conducted on my personal laptop. Processor: Intel Coreffl i7-3612QM CPU @ 2.10GHz 8, Memory Size: 8 GB and GPU: ATI Radeon HD 5450 2GB memory.

4.1 Datasets

Two datasets used for extraction of job and stages features. 41 tests provided by swat itself and the TPCH queries (where tested on CPU configuration only). TPCH queries are data-intensive SQL implemented using Spark-SQL interface, while the SWAT tests are a set of compute intensive algorithms that comes mostly from Apache Spark MLlib such as Neural nets and genetic algorithms.

4.2 Evaluation Metrics

For our model evaluation we used R-squared and mean square error in the evaluation

Metric	Gradient Boosting	SVR
R-Squared	0.76	0.33
MSE	3346.78689743548	11799.042329045044

Table 2: Metric for prediction on stages

Metric	Gradient Boosting	SVR
R-Squared	-4.14147924142	-0.581533211492
MSE	44068.955	25809.03

Table 3: Metric for prediction on jobs

Figure 3: Scatter Plot of Gradient Boosting for Stages

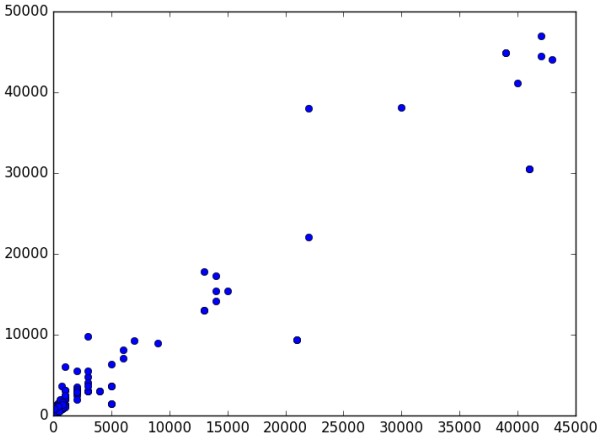
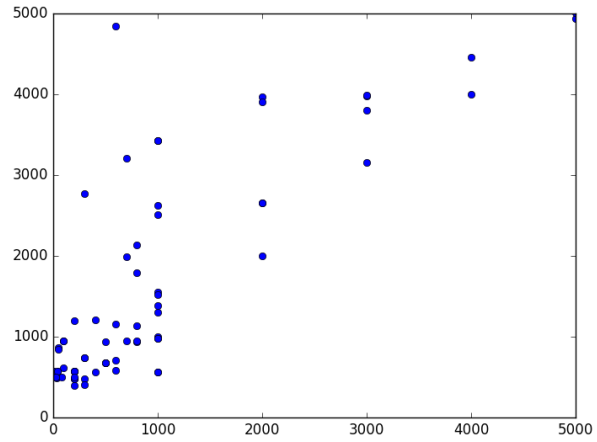


Figure 4: Scatter Plot of SVR for Stages



4.3 Result analysis

As we can see the only decent model in prediction is gradient boosting model for stage prediction, all other models are too bad to be considered for prediction. This is because small datasets used to train jobs while stages has a relatively large datasets for stages. Yet, all these datasets are small and need a special care to their extracted features. Relatively large datasets and different configurations need to be added in order to use more complicated machine learning algorithms and achieve a better accuracy on them.

5 CONCLUSIONS AND FUTURE WORK

Predictive model for Lengthy compute intensive workloads are important to choose the best hardware configuration and also cost

Figure 5: Scatter Plot of Gradient Boosting for Jobs

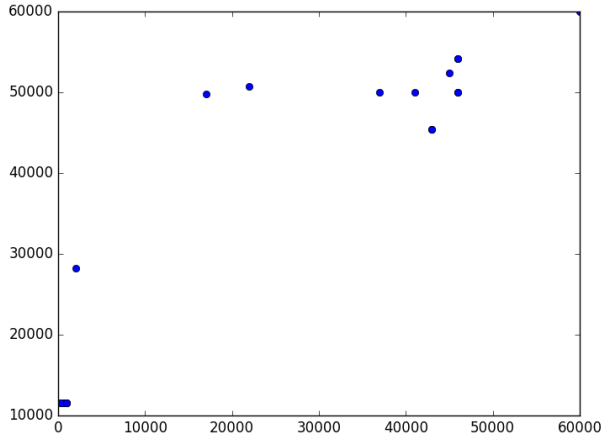
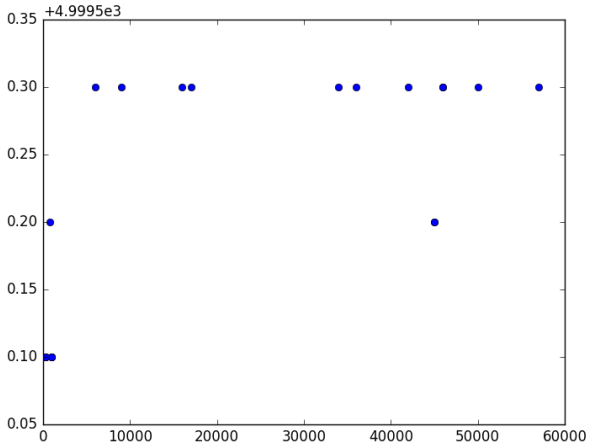


Figure 6: Scatter Plot of SVR for Jobs



estimation of used machines. As we can see machine learning approach for prediction is decent only in predicting stages execution time. while having very bad accuracy for job execution time prediction. Future trends must concentrate on better feature extraction for the job and collecting large amounts of data to prevent overfitting. Also, various test on different machines must be conducted throughly.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, and others. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Archana Ganapathi, Yanpei Chen, Armando Fox, Randy Katz, and David Patterson. 2010. Statistics-driven workload modeling for the cloud. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. IEEE, 87–92.

[3] Aisling OfiDriscoll, Jurate Daugelaite, and Roy D Sleator. 2013. fiBig datafi, Hadoop and cloud computing in genomics. *Journal of biomedical informatics* 46, 5 (2013), 774–781.

[4] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 363–378. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman

[5] Gene Wu, Joseph L Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. 2015. GPGPU performance and power estimation using machine learning. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 564–576.