2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

Diagnosing Performance Bottlenecks in Massive Data Parallel Programs

Vinícius Dias Rubens Moreira Wagner Meira Jr. Dorgival Guedes
Department of Computer Science - Universidade Federal de Minas Gerais
{viniciusvdias, rubens, meira, dorgival}@dcc.ufmg.br

***Abstract*—The increasing amount of data being stored and the variety of applications being proposed recently to make use of those data enabled a whole new generation of parallel programming environments and paradigms. Although most of these novel environments provide abstract programming interfaces and embed several run-time strategies that simplify several typical tasks in parallel and distributed systems, achieving good performance is still a challenge. In this paper we identify some common sources of performance degradation in the Spark programming environment and discuss some diagnosis dimensions that can be used to better understand such degradation. We then describe our experience in the use of those dimensions to drive the identification performance problems, and suggest how their impact may be minimized considering real applications.**

I. Introduction

Data Science emerges as a new paradigm for exploring and getting value from data. It evolved from data mining, machine learning, big data and analytics, among others, in particular, with respect to the models and algorithms employed. On the other hand, data scientists are more pro-active and demanding in terms of task complexity and quality requirements.

An increasing popular strategy for deploying these applications is to parallelize them using "data flow" environments such as Hadoop and, lately, Spark. Those environments provide abstract programming models that simplify the task of the data scientists to describe an algorithm in a way that may be easily deployed in a large number of machines.

In looking for the best performance, those frameworks face several challenges. Finding optimal partitioning is hard because different programs can have different computational costs for the same input and execute user defined functions, which aggravate the issue. Besides that, programs may also require changes of plan and reconfigurations to achieve the best performance over time. Optimizations from a static point of view [1] or when the structure of the data is known [2]–[4] are the most common practices in that sense. However, it is still an open problem to optimize performance when no assumptions can be made about the data or when the execution model imposes limitations to automatic reconfiguration [5].

In this work we present our strategy for performance diagnosis of Spark data intensive applications and instantiate the strategy on applications representing common behaviours in the area. We are aware of one work that systematically study performance bottlenecks in new generation data-parallel frameworks [6] from a general point of view. Instead, we focus on special cases where the common rules do not seem to apply. By identifying some significant dimensions of analysis to guide our experimental evaluation, we are able to identify important factors that can affect the performance of the resulting applications. Based on our analysis we discuss some possible solutions and show opportunities for improvement.

II. Data parallel processing execution model

One can think of a parallel program as a data flow, composed by operators, inputs and outputs. To achieve parallelism, data is partitioned and distributed among compute nodes. The operations to be applied to data in frameworks like MapReduce [7], Dryad [8] and Spark [9] are described using DAGs (**D**irected **A**cyclic **G**raphs). The interpretation given to vertices and edges in those graphs depends on the system, but they always represent the way data flows and is transformed.

In this paper, we chose Spark as the framework to use in our discussions, since it is a modern, widely used system. Spark's engine defines computation DAGs in terms of collections (**R**esilient **D**istributed **D**atasets, RDDs) and transformation operators whose relationships are determined by different types of dependencies [9]. Transformations with *narrow dependencies* represent computations that do not require remote communication and thus can be pipelined (e.g., mapping and filtering). On the other hand, *wide dependencies* mark the need for communication and mark the frontier between *stages of execution* (e.g., reduction and join). Those stages are composed by tasks running the same code and represent opportunities for changing the application parallelism.

III. Performance Diagnosis Dimensions

In this section, we present diagnosis dimensions that will ground our evaluation and provide a common terminology.

Data layout (DL): when programs read their raw input, it is stored in data structures that carry additional semantic information. Primitive and composite types, contiguous arrays and pointers are common ways for making sense of data in memory. Object oriented languages add a layer of abstraction that improve productivity and modularity and for that they are often used in the majority of data parallel frameworks, like Hadoop and Spark. However, the added abstraction comes with increased overhead in memory usage. Furthermore, those frameworks run on top of the Java Virtual Machine (JVM), and the JVM's garbage collector is known to be sensitive to memory layout and access patterns [10]. Thus, a few works

978-1-5090-2453-7/16 $31.00 © 2016 IEEE
DOI 10.1109/CCGrid.2016.81

propose custom alternatives to the traditional garbage collection mechanisms [11], [12]. In fact, even Spark's memory management is moving towards a manual, optimized memory layout [13]. For these reasons, it is worthy to quantify the impact of data layout on performance.

Task placement (TP): considering the established execution model (section II), scaling to the amount of data is straightforward, since data partitioning dictates the parallelism. However, given that applications process giga- or even terabytes of data, at any point of time, hundreds or even thousands of tasks may be ready to be dispatched for execution in the cluster. It is scheduler's role to receive all work requests, pack them into an execution plan and decide which tasks go to which resource. In many applications, every task can have a different computational cost, or process a different amount of data. Thus, the co-allocation of heterogeneous tasks has the potential for creating unexpected performance issues.

Adequate parallelism (AP): achieving the right degree of parallelism does not mean just to tune applications for performance, but also to do so with just *enough* resources at each point during execution. In data parallel systems, the entity responsible for handling parallelism is the partitioner, which organizes the data based on a *number of partitions*, and it must be able to find that adequate parallelism for each execution. Each excution stage may have a different optimal degree of parallelism, and the data shuffling between stages become opportunities to adjust the partitioning accordingly. Therefore, elastic behaviors encountered along application life cycle must be taken into account when tuning an application.

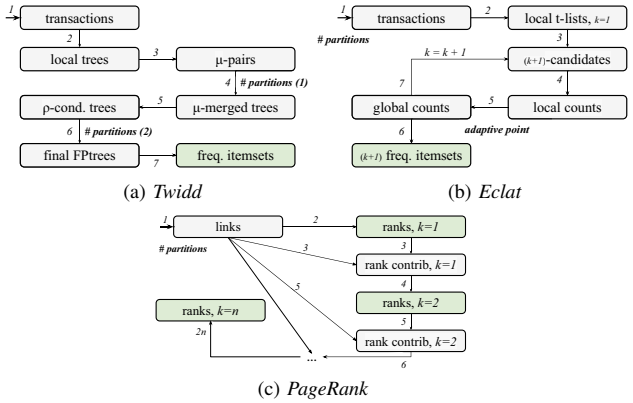
Load balancing (LB): beyond the number of partitions, one must worry about the amount of work assigned to each one of then. For instance, Spark's execution model creates invisible barriers at the beginning of each stage, needed to satisfy data dependencies before proceeding with execution. Assuming applications execute stages sequentially, every imbalance in a stage's tasks lead to resource idleness, which in turn could produce create performance bottlenecks. Such load imbalance might be inherent to the algorithm or, worse, due to bad partitioning, and only a detailed analysis of execution may be necessary to determine how to handle the problem.

IV. Selected applications

We describe next three algorithms that represent common patterns of data parallel algorithms: *(section IV-A)* Twidd; *(section IV-B)* Eclat and *(section IV-C)* PageRank. Next, we evaluate these algorithms w.r.t. the established dimensions.

A. Twidd: non-iterative with complex data structures

Twidd is an implementation of FPGrowth [14] over RDDs. It solves the problem of frequent pattern mining: given transactions, each one composed by sets of items, and a support threshold *minsupp*, find all subsets of items (itemsets) that occur in more than *minsupp* transactions. Most approaches begin by constructing a global table of the frequent 1-itemsets (a.k.a. a subset of length 1). The following description assumes that such table is available in every slave.



(a) *Twidd* (b) *Eclat*

(c) *PageRank*

Fig. 1. Selected applications

An overview of Twidd is shown in Figure 1a. The algorithm starts by reading transactions from the file system (*step 1*). The subsets of transactions serve as input to build local prefix trees, adding items based on their absolute frequency (*step 2*). The next step consists of extracting μ-trees from local trees based on μ-length prefixes (*step 3*). This is an intermediate procedure that mitigate imbalance by replicating FPTree nodes in more than one worker. We map each RDD element (local trees) into potentially several extracted μ-trees (pairs in an one-to-many fashion). The output is an RDD of sub-trees keyed by prefixes. The shuffle occurs (*step 4*) and sub-trees with common prefixes are merged together.

The remaining steps represent FPGrowth's projections. In special, merged trees are conditioned up to level ρ (*step 5*), which results in a new RDD of pairs, similarly to step 3. This is a pre-projection and concludes the Twidd's balancing scheme. Then, a second shuffle guarantees that sub-trees conditioned to the same prefix are merged together (*step 6*). The result is an RDD of final FPTrees, which are independently projected. The output (i.e. the frequent itemsets) is written into HDFS.

We evaluate Twidd in terms of **DL**, **TP** and **LB** due to the fact that its implementation is based on complex data structures (trees) that can expose different behaviors due to garbage collection and heterogeneous task load.

B. Eclat: iterative and irregular

Eclat is another approach for frequent pattern mining. It works with a vertical database layout for fast candidate counting by intersecting inverted lists of transactions. For simplicity, the following description also assumes that the global table of frequent 1-itemsets is available.

Figure 1b illustrates our implementation. The procedure starts by reading the transactions from HDFS (*step 1*). Next, we perform a database verticalization as required for Eclat (*step 2*). However, each RDD partition is verticalized independently, which avoids the creation of huge t-lists [15].

The remaining frequent itemsets are found hierarchically and iteratively. We generate $(k+1)$-itemset candidates for each local partition by intersecting the current t-lists (*step 3*). Local counts are extracted for the new candidates (*step 4*) and aggregated globally (*step 5*). These frequencies have two

purposes: output frequent $(k+1)$-itemsets (*step 6*) and filter infrequent candidates to build new t-lists for the next iteration (*step 7*). The process continues until we reach a level where no frequent itemset was found.

We evaluate Eclat in terms of **DL** due its implementation based on simple data structures. Also, Eclat's trade-off between the number of local databases (partitions) and the degree of t-list replication along with its hierarchical and combinatorial nature exhibit potential for **AP** evaluation.

C. PageRank: iterative and regular

PageRank is a link analysis algorithm that computes the relative importance of nodes in a network. Figure 1c illustrates the algorithm. The links are loaded from storage and aggregated in the form of adjacency lists (*step 1*). The resulting RDD is cached for future reuse. Then, we assign the initial ranks (initially 1.0) to the links (*step 2*). Current ranks are joined with corresponding adjacency lists and produce rank contributions (*step 3*). Thus, a node with rank r and n neighbors would share $\frac{r}{n}$ of its own rank with each neighbor. We update ranks by summing individual contributions for each node. The result is a new RDD of ranks. The process continues for a number of iterations.

We evaluate it in terms of **AP**, since it may expose opportunities for optimization of iterative and regular applications.

V. Evaluation

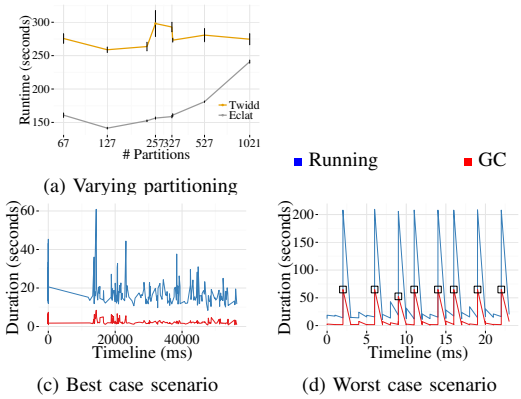
Our experimental environment is a cluster composed of 9 machines, each containing a quad-core Intel Xeon X3440 with hyperthreading and 8 MB cache, 16 GB RAM, a 1 TB 7200 RPM SATA disk, running 64-bit Linux 3.2.0. The nodes are connected with Gigabit Ethernet. The cluster is configured with Spark v1.5.1 and Hadoop/HDFS v2.5.0. We used two real world data sets in our evaluation: a set of Twitter posts, containing tweets crawled using the Twitter API ($7.9GB$, $233mi$ transactions and $64mi$ distinct items); and a Google+ graph representing users/nodes and friendships/edges [16] ($9.3GB$, $35mi$ nodes and $575mi$ edges). The former dataset is used in Twidd/Eclat's evaluation while the latter in PageRank's.

A. Task placement

Initially we observed the behavior of Twidd and Eclat under several degrees of parallelism. Figure 2a shows the results. In Twidd we vary the degree of parallelism of *# partitions(1)*. In Eclat we vary the input parallelism, i.e., to *# partitions*.

Partitioning in Eclat is what one would expect, in which there is a sweet spot that represents the best trade-off for parallelism., as for Twidd not so much. Twidd's run-time behaves similarly to Eclat's in the beginning. However, at the range $[227, 331]$ we note some interesting phenomena: run-time increases with 257 and 327 partitions. Also, we observe a high variation in the results with 257 partitions.

Next we isolate the stage causing high variation to observe its tasks over time as a function of respective running times in best/worst case scenarios (Figures 2c and 2d). Note that running times stay between 10 and 30 sec in most tasks. As



(a) Varying partitioning

(c) Best case scenario

(d) Worst case scenario

Fig. 2. Detailed analysis of Twidd's 5th stage

for the worst case scenario we plot the same timeline, but zoomed in on the tasks that have caused the high increase in run-time (72 of 257). Comparing the two scenarios we note that not even the most costly outlier of the best case came close to the upper bound caused by 8 tasks in the worst case (≈200 sec). We finish our diagnosis by observing that those stragglers were scheduled nearly at the same time (delta 30ms) in the same executor 6.

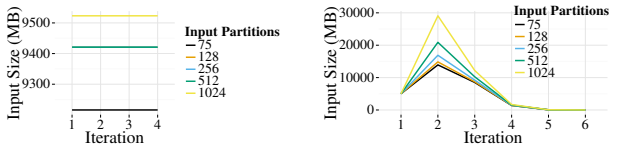
Highlights related to **TP** *and* **DL***:* irregular applications are specially sensitive to parallelism and task placement. In Twidd, we have identified cases where co-allocation of heavy tasks on executors running slow due to GC pauses could bring significant performance degradation to the overall execution. That behaviour is tighly coupled with the complexity of data structures employed by Twidd. In fact, this is the reason in which Eclat presents such predictable behavior w.r.t. varying the degree of parallelism. Therefore, data parallel schedulers could benefit of GC-awareness from its worker nodes.

B. Adaptive execution

A stage of execution in Spark can be described in terms of: its function; data inputs and shuffle read (i.e. inputs) and data outputs and shuffle writes (i.e. outputs). Many machine learning and graph processing algorithms are based on iterative and/or converging approaches. When translated into DAGs, these techniques expose opportunities for re-execution optimization. By tying up program's logic with the data it consumes/produces, one can classify some groups of stages into: (P_{eq}) *Equal*, which executes the same function with the same input/outputs; and (P_{sim}) *Similar*, which executes the same functions with different inputs/outputs.

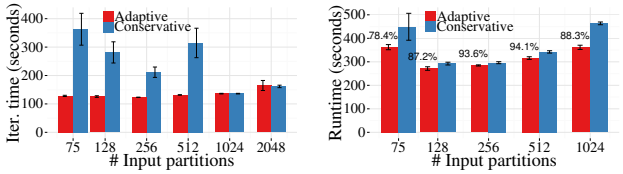
Algorithms like PageRank produces the same communication pattern every iteration: graph nodes share rank contributions. Therefore, iterations stages are P_{eq}. Figure 3a shows the constant progression of PageRank's stage inputs as a function of iterations. We also note algorithms like Eclat that re-execute stage functions several times but in different contexts. Threfore, the problem has the potential for growing/shrinking over time (Figure 3b). This is the case of P_{sim} stages.

Optimizing P_{eq}: given that we find the right amount of parallelism from earlier iterations, one could use that knowledge for later iterations to speedup the execution. We



(a) PageRank (b) Eclat

Fig. 3. Input size over iterations.



(a) P_{eq}: 1024 partitions represent our baseline for optimization.

(b) P_{sim}: the degree of parallelism must adapt to the elastically.

Fig. 4. Optimizing P_{eq} and P_{sim}

applied that optimization to PageRank. Figure 4a shows the results. Initially, we ran the algorithm with different degrees of parallelism (*conservative* approach). Then, after identifying that 1024 partitions would speedup the reductions, we re-ran the tests in the other configurations applying this knowledge to the reduction step. We refer to that as adaptive because regardless of input partitioning (which users have static control), we are able to optimize the re-executed step according to its inherent cost. Note that we improved overall performance in the first four configurations.

Optimizing P_{sim}: when the problem size varies along the iterations, the problem elasticity must be addressed properly. The application should be able to estimate new configurations based on the common knowledge of re-executions. We validate this idea by creating a heuristic applied to adapt the Eclat's parallelism. We observe the selectivity factor of the first operator before the shuffle took place, i.e., $selec = \frac{input}{shuffleWrite}$. This factor is used to tune the parallelism for the reduction in the first iteration. We set the number of reducer partitions to $nparts = \max(\lceil \frac{inputPartitions}{selec} \rceil, totalCores)$. The remaining iterations are tuned proportionally to $shuffleWrite$ with $nparts$ of the first iteration. Thus, the parallelism would follow the elasticity along the iterations.

Figure 4b shows our results. The adaptive heuristic is specially efficient on fixing under and overestimations (75, 1024 input partitions). Adaptive execution also allows a more conscious use of resources. Note that the number of tasks launched is reduced by at least 78.4% in all configurations.

Highlights related to **AP***:* system optimizers could adapt the execution according to previously observed communication patterns, leveraging that knowledge to improve performance and encourage using the right amount of resources.

C. Load balancing

We investigated the 4th and 5th stages of Twidd w.r.t. load balancing. Due to the space limitations we only present our methodology and the most relevant findings. One can diagnose imbalance by observing the distribution of some important metrics regarding each stage: *run-time* (**rt**); *shuffle read* (**sr**) and *number of records* (**nr**). If we get high correlation between **rt** and **sr** along with less skewness in **nr**, then may be the case of imbalance explained by tasks with variable costs (4th stage) and thus, (hash)-partitioning is being naive. On the other hand, symmetrical distributions on **sr** and **nr** along with high skewness in **rt** indicate inherent imbalance (5th stage).

Highlights related to **LB***:* load balancing may arise from an inherent load distribution or from naive partitioning. Also if the number of records is not a good estimate of tasks' cost then advanced instrumentation may be the only reliable solution.

VI. Conclusion

Understanding elements affecting the performance of applications in massive data parallel programs may be a difficult task due multiple challenges posed by the available frameworks. We have diagnosed bottlenecks in data parallel systems over performance dimensions that reflect those challenges. Our highlights show that sources of inefficiency can be obvious or not. Thus, it is important to establish methodologies for evaluation of data parallel systems, which is key to identify opportunities for performance optimization.

Acknowledgements

This work was partially funded by Fapemig, CNPq, CAPES, and by projects InWeb (MCT/CNPq 573871/2008-6), MASWeb (FAPEMIG-PRONEX APQ-01400-14), and EUBra-BIGSEA (H2020-EU.2.1.1 690116, Brazil/MCTI/RNP GA-000650/04).

References

[1] H. Herodotou *et al.*, "Starfish: A self-tuning system for big data analytics," in *In CIDR*, 2011, pp. 261–272.

[2] R. Chaiken *et al.*, "Scope: Easy and efficient parallel processing of massive data sets," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1265–1276, Aug. 2008.

[3] Q. Ke, M. Isard, and Y. Yu, "Optimus: A dynamic rewriting framework for data-parallel execution plans," *Proceedings of the 8th ACM European Conference ...*, pp. 15–28, 2013.

[4] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *NSDI'12*, 2012.

[5] M. Zaharia, "Adaptive execution in spark," 2015. [Online]. Available: https://issues.apache.org/jira/browse/SPARK-9850

[6] K. Ousterhout *et al.*, "Making sense of performance in data analytics frameworks," in *NSDI 15*. Oakland, CA: USENIX Association, May 2015, pp. 293–307.

[7] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[8] M. Isard *et al.*, "Dryad: Distributed data-parallel programs from sequential building blocks," in *EuroSys '07*, 2007.

[9] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI 12*. San Jose, CA: USENIX, 2012, pp. 15–28.

[10] K. Nguyen *et al.*, "Facade: A compiler and runtime for (almost) object-bounded big data applications," in *ASPLOS '15*, 2015.

[11] I. Gog *et al.*, "Broom: Sweeping out garbage collection from big data systems," in *HotOS XV*. USENIX Association, May 2015.

[12] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz, "Trash day: Coordinating garbage collection in distributed systems," in *HotOS XV*. USENIX Association, May 2015.

[13] R. Xin, "Project tungsten," 2015. [Online]. Available: https://issues.apache.org/jira/browse/SPARK-7075

[14] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *SIGMOD Rec.*, vol. 29, no. 2, pp. 1–12, May 2000.

[15] A. Veloso, W. Meira, Jr., R. Ferreira, D. Guedes, and S. Parthasarathy, "Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining," in *PKDD '04*, 2004.

[16] G. Magno *et al.*, "New kid on the block: Exploring the google+ social graph," in *IMC '12*, 2012.