Massively Parallel Processing of Convolutional Neural Networks with Spark: The MareNostrum Experience

(authorship information omitted for blind review)

***Abstract*—Deployment of a distributed deep learning technology stack on a large parallel system is a very complex process, involving the integration and configuration of several layers of both, general-purpose and custom software. The details of such kind of deployments are rarely described in the literature. This paper presents the experiences observed during the deployment of a technology stack to enable deep learning workloads on MareNostrum, a petascale supercomputer. The components of the chosen layered architecure are described and the performance and scalability of the resulting system is evaluated. This is followed by a discussion about the impact of different configurations including parallelism, storage and networking alternatives, and other aspects related to the execution of deep learning workloads on a traditional HPC setup. The derived conclusions should be useful to guide similarly complex deployments in the future.**

I. Introduction

Traditional HPC (High-Performance Computing) systems are designed according to the compute-centric paradigm, with focus on computing power, and the goal to process as many floating-point operations per second as possible. However, the growing importance of data-intensive applications is currently pushing the transition of many computing facilities into a data-centric paradigm, for which the variable to maximize is the amount of data, measured in records or bytes, processed per second to perform data analysis. This paradigm shift poses a dilemma to the managers of traditional HPC facilities, who have to choose between deploying dedicated systems for data analytics or to evolve their existing infrastructure to meet the new demands. The work described in this paper explores the second option, adapting an existing HPC setup to enable deep learning workloads on a real-world, petascale, HPC setup, the MareNostrum supercomputer. The work details the design and evaluation of a framework to distributedly train deep neural networks. The core of the framework is based on the usage of Apache Spark [10], which isolates the application layer (CNNs training with DL4J) from the infraestructure (Spark4MN and the MareNostrum supercomputer). Our framework integrates these four elements (DL4J, Spark, Spark4MN and MareNostrum) enabling to efficiently train deep neural networks over thousands of cores. Apart from design and deployment, the challenge is scalability and proper configuration. Simply running on many CPUs may yield poor benefits or even degraded performance due to overheads. We deal with this issue and we aim to make the first step towards systematic analysis of the several parameters and optimized configuration. More specifically, we evaluate the behavior of a representative deep learning application, image recognition. We discuss the impact of several configuration parameters related to massive parallelism and we provide insights into how the job configuration on a HPC compute-centric facility can be optimized to efficiently run this kind of workloads.

II. Related Work

[TODO: RUBEN, half a column]

Many works has addressed the efficient configuration of MapReduce environments (e.g. [1], [2]). Regarding HPC setups, [4] describes a framework to enable Hadoop workloads on a Cray X-series supercomputer. Regarding Spark, in [8], the performance of Spark on an HPC setup is investigated. This work studies the impact of storage architecture, locality-oriented scheduling and emerging storage devices. In [3], the authors compare the performance of traditional HPC setups against Hadoop-like frameworks over clusters of commodity hardware with respect to the processing of data-intensive workloads.

[TODO: image recognition] [TODO: deep learning on HPC works] [TODO: deep learning on Spark works] [TODO: mencionar que la cosa va mejor con GPUs pero que tenemos casi 50K CPUs...] [TODO: mencionar unsupervised learning and GANs for future work]

autocitas:

[6] = multimedia big data [7] = UGC

III. A framework for training deep neural networks on MareNostrum

The goal of the proposed framework is to be able to take profit of the computation resources provided by MareNostrum (almost 50K cores and more than 100TB of aggregated RAM) for training deep neural networks. In order to isolate data-intensive applications from the particularities of the supercomputer, Apache Spark is normally used as an intermediate layer (not only in MareNostrum, [8] does the same on a Cray X-series supercomputer). The deployment of Spark-enabled clusters over MareNostrum is not trivial, it has required the development of a specific interoperability layer that we call Spark4MN, which will be explained later. On top of this stack (Marenostrum, Spark4MN and Spark) we place a deep learning specific layer, DL4J.

DL4J [TODO: Leonel]...

In order to evaluate the performance and scalability of the proposed framework, we have experimented with different workloads (datasets and network architectures) and different framework setups (number of nodes, parallelism configuration, etc.). All the selected workloads are related to image datasets and image-related applications (e.g. image recoginition).

[TODO: Leonel, explicar un poquito mÃ¡s las pruebas que hacen]

Through the following sections we explain the different components of the framework with more detail. Then, we discuss the performed experiments and the obtained results, aiming to shed light onto the parameters that have the biggest impact and their effective configuration.

IV. Convolutional Neural Networks

V. DL4J

DLFJ (or Deeplearning4j) is a computing framework written for Java with wide support for deep learning algorithms. DL4J provides distributed parallel versions (both for GPUs and CPUs) of the algorithms that integrate with Apache Hadoop and Spark. In order to achieve distributed network training over Spark, DL4J applies parameter averaging. Training data is divided into a number of splits. Each split is in turn divided into a number of minibatches. At each iteration all the minibatches of one data split are processed in parallel by the available Spark workers. All the resulting parameter updates are averaged and returned to the Spark master. There are several parameters that must be adjusted to optimize training time. These include, but are not limited to, minibach size (number of examples for each parameter update in each worker), averaging frequency (too low averaging periods may imply too networking overhead), prefetching (how many minibatches a worker must prefetch to avoid waiting for the data to be loaded), repartitioning strategy (when and how to repartition data to keep the partitions balanced and the proper level of parallelism, and data locality.

In addition to properly configure DL4J, in some cases it's also necessary to adjust some Spark default parameters to fit the particularities of a deep learning workload. Deep learning is computationally intensive, and hence the amount of computation per data partition is relatively high. Spark default locality level (when does it make sense to await for a executor closer to the data to become free) needs to be set to zero, as computation time will allways outweigh any network transfer time.

[TODO: Diagrama]

VI. Apache Spark

As mentioned before, Apache Spark is the key component of the proposed framework. Spark is a distributed system for processing data-intensive workloads. It excels in an efficient memory usage, outperforming Hadoop for many applications [9]. Spark is being used to execute big data workloads on the MareNostrum supercomputer, isolating the applications from the particularities of this HPC infraestructure. Spark is designed to avoid the file system as much as possible, retaining most data resident in distributed memory across phases in the same job. Such memory-resident feature stands to benefit many applications, such as machine learning or clustering, that require extensive reuse of results across multiple iterations. Essentially, Spark is an implementation of the so-called Resilient Distributed Dataset (RDD) abstraction, which hides the details of distribution and fault-tolerance for large collections of items.

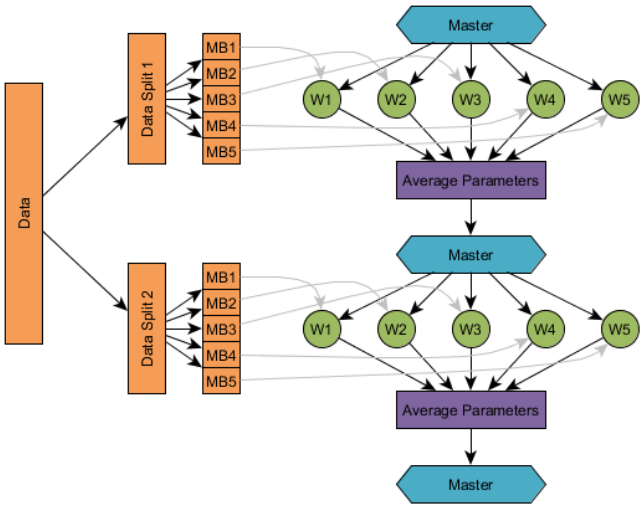

Fig. 1: Parameter averaging in DL4J over Spark

RDDs provide an interface based on coarse-grained *transformations* (e.g., *map, filter* and *join*) that apply the same operation to many data items. Spark computes RDDs lazily the first time they are used in an *action*, so that it can pipeline transformations; *actions* are operations that return a value to the application or export data to a storage system. In our work, we focus on cases where the aggregate memory can hold the entire input RDD in main memory, as typically happens in any HPC infrastructure.

Spark attempts to include all the transformations that can be pipelined in a single stage to boost performance. Between different stages, it is necessary to "shuffle" the data. The shuffling of intermediate data constitutes the major performance bottleneck of all MapReduce implementations and its descendants, including Spark. When a shuffle operation is encountered, Spark first flushes in-memory output from the previous stage to the storage system (storing phase), possibly storing also to disk if allocated memory is insufficient; then it transfers the intermediate data across the network (shuffling phase).

[TODO: Particularidades de Spark en el caso de deep learning workloads]

VII. The Spark4MN Framework

The MareNostrum supercomputer is accessed through an IBM LSF Platform workload manager. In order to be able to deploy Spark clusters over MareNostrum, we employ an intermediate layer called Spark4MN [5]. Spark4MN is also in charge to manage the deployment of any additional resource Spark needs, such as a service-based distributed file

system (DFS) like HDFS. Essentially, Spark4MN is a collection of *bash* scripts with three user commands (*spark4mn*, *spark4mn_benchmark* and *spark4mn_plot*). *spark4mn* is the base command, which deploys all the Spark cluster's services, and executes the user applications. *spark4mn_benchmark* is a test automation command to execute the same user application with a series of different hardware configurations. All the metric files generated by a benchmark are finally gathered by *spark4mn_plot*.

a) Cluster setup and Spark application submission: Spark4MN scripts read a configuration file, describing the application and the Spark cluster configuration, provided by the user (see below) and submits one or more jobs to the MareNostrum workload manager. Once the cluster's job scheduler chooses a Spark4MN job to be executed, an exclusive number of cluster's nodes are reserved for the Spark cluster and (if requested) for the DFS (e.g. HDFS) cluster (may be the same nodes, depending on the configuration). After the resource allocation procedure, Spark4MN starts the different services. If a DFS is requested, its master service (e.g. the HDFS *namenode* service) is executed first. Then, all the DFS worker services (e.g. the HDFS *datanode* services) are launched and connected to the master. Once the DFS cluster has been setup the Spark setup is done in the same way (wait for master to be ready, start workers). In Spark4MN, the Spark master corresponds to the *standalone* Spark manager, and workers are Spark worker services, where the Spark executors are received and launched. The cluster startup requires about 12 seconds. This is independent of the size of the cluster (the number of nodes). Since real world applications (e.g. PubMed article processing) may run for dozens of minutes, this constitutes an acceptable overhead. Each application is executed via *spark-submit* calls. During each Spark job execution, intermediate data is produced, e.g., due to shuffling. Such data are stored on the local disks and not on DFS by default (as in [4], this yields the best performance). Finally, Spark timeouts are automatically configured to the maximum duration of the job, as set by the user.

VIII. MareNostrum supercomputer

[TODO: Maybe mention the version 4?]

MareNostrum is the Spanish Tier-0 supercomputer provided by BSC. It is an IBM System X iDataplex based on Intel Sandy Bridge EP processors at 2.6 GHz (two 8-core Intel Xeon processors E5-2670 per machine), 2 GB/core (32 GB/node) and around 500 GB of local disk (IBM 500 GB 7.2K 6Gbps NL SATA 3.5). Currently the supercomputer consists of 48896 Intel Sandy Bridge cores in 3056 JS21 nodes, and 84 Xeon Phi 5110P in 42 nodes (not used in this work), with more than 104.6 TB of main memory and 2 PB of GPFS (General Parallel File System) disk storage. More specifically, GPFS provides 1.9 PB for user data storage, 33.5 TB for metadata storage (inodes and internal filesystem data) and total aggregated performance of 15GB/s. The GPFS filesystems are configured and optimized to be mounted on 3000 nodes. All compute nodes are interconnected through an Infiniband FDR10 network, with a non-blocking fat tree network topology. In addition to the 40 Gb/s Infiniband, 1 Gb/s full duplex Ethernet is in place. With the last upgrade, MareNostrum has a peak performance of 1.1 Petaflops.

IX. Experiments and Results

[TODO: LEONEL, 1 and a half column]

The main goal of the experiments is to evaluate the scalability properties of the proposed framework applied to the selected workloads. To this end, we use

[TODO: Describe workloads = datasets, netwoks, etc.]

The cluster sizes range from 8 cores up to 1024 (i.e., 64 machines)...

[TODO: Describe configurations.]

We have submitted and tested several hundreds of jobs to MareNostrum, but we describe only the results that are of significance. Our runs include an extensive set of configurations; for brevity, when those parameters were shown to be either irrelevant or to have negligible effect, we use default values. Each experimental configuration was repeated at least 5 times. Unless otherwise stated, we report median values in seconds.

[TODO: Si hay algÃºn resultado explicarlo y poner el plot. Si no nos limitamos a explicar los experimentos realizados.]

X. Conclusions

The research work presented in this paper..... [TODO: Ruben]

Acknowledgements

This work is partially supported by the Spanish Ministry of Economy and Competitivity under contract TIN2015-65316-P and by the SGR programme (2014-SGR-1051) of the Catalan Government.

,

References

[1] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SOCC*, pages 18:1–18:14, 2011.

[2] Botong Huang, Shivnath Babu, and Jun Yang. Cumulon: optimizing statistical data analysis in the cloud. In *SIGMOD*, pages 1–12, 2013.

[3] Shantenu Jha, Judy Qiu, André Luckow, Pradeep Kumar Mantha, and Geoffrey Fox. A tale of two data-intensive paradigms: Applications, abstractions, and architectures. In *IEEE Int. Congress on Big Data*, pages 645–652, 2014.

[4] Scott Michael, Abhinav Thota, and Robert Henschel. Hpchadoop: A framework to run hadoop on cray x-series supercomputers. In *Cray USer Group (CUG)*, 2014.

[5] Rubén Tous, Anastasios Gounaris, Carlos Tripiana, Jordi Torres, Sergi Girona, Eduard Ayguadé, Jesús Labarta, Yolanda Becerra, David Carrera, and Mateo Valero. Spark deployment and performance evaluation on the marenostrum supercomputer. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 299–306, 2015.

[6] Ruben Tous, Jordi Torres, and Eduard Ayguade. Multimedia big data computing for in-depth event analysis. In *BigMM*, pages 144–147. IEEE, 2015.

[7] Ruben Tous, Otto Wust, Mauro Gomez, Jonatan Poveda, Marc Elena, Jordi Torres, Mouna Makni, and Eduard Ayguade. User-generated content curation with deep convolutional neural networks. IEEE, 2016.

[8] Yandong Wang, R. Goldstone, Weikuan Yu, and Teng Wang. Characterization and optimization of memory-resident mapreduce on hpc systems. In *IPDPS*, pages 799 – 808, 2014.

[9] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. HotCloud'10.