Distributed Training of Deep Neural Networks with DL4J and Spark on the MareNostrum Supercomputer

(authorship information omitted for blind review)

***Abstract*—In this paper we describe the design and evaluation of a framework to enable deep learning workloads on MareNostrum, a petascale supercomputer. We present the design of the framework and evaluate the scalability of the system. We examine the impact of different configurations including parallelism, storage and networking alternatives, and we discuss several aspects in executing deep learning workloads on a computing system that is based on the compute-centric paradigm. We derive conclusions to facilitate systematic and optimized methodologies for fine-tuning this kind of workloads on large clusters.**

I. Introduction

The work described in this paper describes the design and evaluation of a framework to enable deep learning workloads on a real-world, petascale, HPC setup, the MareNostrum supercomputer. The core of the framework is based on the usage of Apache Spark [3], 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 of 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]

III. Global Framework Design

[TODO: RUBEN, half a column, maybe a diagram]

IV. DL4J

[TODO: LEONEL, one column]

V. Apache Spark

Apache Spark is an open-source cluster computing framework. Memory usage is the key aspect of Spark and the main reason that it outperforms Hadoop for many applications [2]. 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.

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). Due to shuffling overhead, when employing m more machines, it is very common to achieve speed-ups considerably smaller than m; in general, shuffling overhead is proportional to the number of machine pairs, i.e., $O(m^2)$.

VI. 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. 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 [1], this yields the best performance). Finally, Spark timeouts are automatically configured to the maximum duration of the job, as set by the user.

VII. 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.

VIII. 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.]

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

[2] 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.

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