Massively Parallel Processing of Deep 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 architecture 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

Over the past several years, deep neural networks (DNNs) have proven to be an incredibly effective tool for a variety of problems, from computer vision, speech recognition or natural language processing. Their number of parameters and complexity, and the size of the training datasets, have quickly grown, leading to be a first class workload for HPC (High-Performance Computing) infrastructures. However, enabling deep learning workloads 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 a real-world, petascale, HPC setup, the MareNostrum supercomputer.

The goal of the deployment 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 DNNs. Nowadays, the usage of GPUs has proven to be the more efficient alternative to train neural networks, speeding up common operations such as large matrix computations [1], [6]. As their price, performance and energy efficiency improves, GPUs are gaining ground in HPC (both in special-purpose systems and in hybrid general-purpose supercomputers). However, there are still many systems, such as MareNostrum, that continue to use conventional CPUs, as they provide a better overall performance-energy-cost trade-off when used for heterogeneous compute-intensive workloads.

The key element of the deployed layered architecure is Apache Spark [14]. In order to isolate machine-learning applications from the particularities of MareNostrum, Spark is usually used as an intermediate layer (not only in MareNostrum, [11] 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, that is written in Java and has a direct integration with Spark, enables distributed training of deep neural networks through a synchronous data parallelism method.

These four elements (DL4J, Spark, Spark4MN and MareNostrum) have been integrated enabling to efficiently train deep neural networks over thousands of cores. Apart from the deployment details, the challenge is scalability and proper configuration. Simply running on many cores 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.

In order to evaluate the performance and scalability of the proposed software stack on MareNostrum, we have experimented with different workloads and different deployment setups (number of nodes, parallelism configuration, etc.). Through the following sections we explain the different components of the deployment in 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. We provide insights into how the job configuration on a traditional HPC setup can be optimized to efficiently run this kind of workloads. The derived conclusions should be useful to guide similarly complex deployments in the future.

II. Related Work

Several works have addressed the execution of deep learning workloads on large specific purpose clusters (e.g. [12]), usually involving nodes equipped with GPUs. Deployments over general-purpose HPC systems are less common, and their details are rarely described in the literature. The work described in [3] analyzes the main bottlenecks of synchronous distributed DNNs SGD-based training. The authors conclude that the issue is quickly turning into a vastly communication bound problem which is severely limiting the scalability in most practical scenarios. In [5] authors present a Caffe-based approach to execute deep learning workloads on a contemporary HPC system equipped with Xeon-Phi nodes. They use the Intel distribution of Caffe, that improves Caffe performance when running on CPUs. Authors report to be able to scale the training of a model up to almost 10K nodes, demonstrating

that deep learning can be optimized and scaled effectively on many-core, HPC systems.

Recent papers that have attempted to scale synchronous deep learning have stopped at a few hundred nodes [21], [20], [24], with the scalability depending on the computation to communication ratio, the speed of the hardware and the quality of the interconnect. Aside from communication there are other factors that limit synchronous scaling:

Our approach attempts to overcome the complexity of a direct deployment such as this by relying on an intermediate layer, Apache Spark. In addition to reduce the deployment costs, our approach enables a systematic tuning of the different configuration parameters, both at application level and at infrastructure level. Other works have followed a similar approach but for other kind of workloads. In [7], authors describe a framework to enable Hadoop workloads on a Cray X-series supercomputer. In [11], 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 [2], 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.

III. Deep Neural Networks

Deep neural networks (DNNs) are layered compositional models that enable learning representations of data with multiple levels of abstraction. State-of-the-art DNNs include many variants, specialized in different domains (convolutional deep neural networks, recurrent neural networks, etc.). DNNs are usually trained by using iterative, gradient-based optimizers (typically minibatch SGD) that drive a non-convex cost function to a local minima. In every iteration step we use information about the gradient ∇E at the current point. In iteration step $[t+1]$ the weight update $\Delta w[t]$ is determined by taking a step (γ is the learning rate) into the direction of the negative gradient at position $w[t]$ such that (in the case of stochastic training):

$$\Delta w[t] = -\gamma \frac{\partial E_n}{\partial w[t]} \tag{1}$$

State-of-the-art networks have a huge number of weights W and the core computation in their training is dominated by dense linear algebra. Usually, in order to improve the efficiency, the training dataset is splitted into mini-batches of size B (typically chosen between 1 and a few hundreds) and the model is only updated (one iteration) after accumulating the gradients of all the training samples within a mini-batch.

DNNs training on a single node involves several software and hardware layers. At the top of the stack there is normally a deep learning framework such as DL4J, TensorFlow, Torch, etc. (there may be even an upper layer such as Keras). Below, the framework relies on an underlying numerical library such as NVIDIA's cuDNN or Intel's MKL. Finally, the models are usually trained on NVIDIA GPUs or Intel's Xeon Phi processors.

When trained on multiple nodes, one can apply data parallelism (distributing training samples among nodes) and/or model parallelism (distributing model parameters among nodes). In our deployment we only apply data parallelism, as it is the only supported by DL4J. The B training samples within a min-batch are splitted into n equal sized sets of size b (with $b = B/n$). The resulting mini-batch-splits are then fed to n nodes holding a complete copy of the model. The results (gradients) off all nodes are then accumulated and used to update the model.

While DL4J limits us to perform this process synchronously (awaiting all the workers to finish before updating the model), it could be also performed asynchronously (allowing model updates with just a part of nodes results). Asynchronous data parallelism can potentially gain higher throughput, but depending on the infrastructure status we can have the *stale gradient problem*. By the time a slow worker has finished its calculations based on a given state of the model, the model may have been updated a number of times and the outdated update may have a negative impact.

IV. DL4J

DL4J (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 performs a version of the synchronous data parallelism mechanism called parameter averaging. Instead of transferring gradients to the master, the nodes perform first a local model update and then they transfer the resulting weights to the master, where they are averaged. With respect to generic parameter averaging, in DL4J the Spark driver and reduction operations take the place of the parameter server (see Figure 1) .

There are several parameters that must be adjusted to optimize training time. These include, but are not limited to, mini-batch-split size, averaging frequency (too low averaging periods may imply too networking overhead), prefetching (how many mini-batch-splits 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.

V. 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 [13]. Spark is being used to execute big data workloads on the MareNostrum supercomputer, isolating the applications from the particularities of this HPC infrastructure. 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

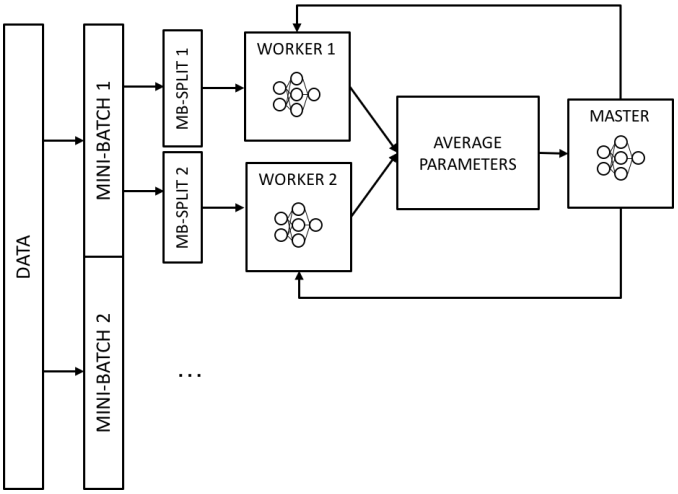

Fig. 1: Parameter averaging in DL4J over Spark

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. 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).

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 [10]. 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 that deploy the Spark cluster's services, and executes the user applications. Spark4MN scripts read a configuration file, describing the application and the Spark cluster configuration, and submit 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. 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). 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 [7], 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

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, 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

The main goal of the experiments is to evaluate the scalability properties of the proposed deployment. To this end, we have experimented with different workloads and different deployment setups. Regarding the benchmarking workloads, we have chosen two widely used convolutional networks, AlexNet [4] and GoogLeNet [9]. Both networks have been used in other state-of-the-art works and let us compare our results with others. While AlexNet implements a rather shallow network with many parameters, GoogLeNet is a very deep network with many convolutional layers. We apply both networks to dataset of the ImageNet [8] visual recognition challenge. Regarding the deployment setup, we have tested different values for the number of nodes, the number of Spark workers per node, the Spark data partition size, the DL4J minibach-split size, the DL4J averaging frequency, prefetching and repartitioning strategy. The results of our evaluation show that DL4J and Spark are able to scale deep learning workloads over MareNostrum. However, the effective scaling stops above 32 nodes with the best configurations. This limitation agrees with the results reported in [3], that studies the theoretic constraints of synchronous data parallelism for DNNs training. The main bottleneck of the synchronous approach is the computation to communication ratio. The synchronous parallelization of DNN training requires the communication of the model w_t and the computed gradients Δw_t between all nodes in every

iteration t. Since w has to be synchronous in all nodes and Δw_{t+1} can not be computed before w_t is available, the entire communication has to be completed before the next iteration $t+1$. The problem is that w and Δw have the size of all weights in the neural network, which can be hundreds of megabytes. The compute times per iteration are rather low and decrease when scaling to more nodes. Depending on the model size and layout, the training problem becomes communication bound after scaling to only few nodes. Shallow networks with many neurons per layer (like AlexNet) scale worse than deep networks with less neurons (like GoogLeNet) where longer compute times meet smaller model sizes. As model sizes are increasing much faster than the available network bandwidth, the communication overhead remains an unsolved problem.

A second problem of the synchronous approach is that nodes process mini-batch-splits instead of mini-batches, and the size b of these splits depends on the number of nodes n. If b is too small, there will be a negative impact on the inner parallel computation (within the node), specially in the case of the FullyConnected (FC) layers. One solution would be to increase the mini-batch size in parallel to the number of nodes, but large batch sizes have been shown to cause slowdown in convergence and degrade the generalization properties of the trained model [5]. A third problem is stragglers. The duration of the iteration depends on the slowest node. This effect gets worse with scale.

Asynchronous parallelization, not possible currently with the deployed software stack, would solve these problems but, as mentioned before, has the *stale gradient problem*. Some recent works like [5] propose a hybrid approach in which synchronous parallelism just takes place within groups of nodes.

IX. Conclusions

The research work presented in this paper explores the feasibility and efficiency of using Apache Spark and DL4J for deploying deep learning workloads over a real-world, petascale, HPC setup, such as MareNostrum. To this end, we have designed a layered architecture consisting in both, general-purpose (Spark and DL4J) and custom components (Spark4MN). We have evaluated the deployment by training AlexNet and GoogLeNet over the ImageNet dataset. We have tested different deployment setups (number of nodes, number of Spark workers per node, data partition size, mini-batch size, mini-bach-split size, averaging frequency, prefetching and repartitioning strategy).

We conclude that it is feasible to rely on Apache Spark to deploy deep learning workloads over a traditional HPC setup. This approach minimizes deployment costs and enables a systematic tuning of the different configuration parameters, both at application level and at infrastructure level. However, the effective scaling is strongly limited by the synchronous parallelism approach applied by DL4J. Problems such as the communication overhead, mini-batch-split size and stragglers degradate the scalability beyond 32 nodes. In order to overcome this limitation it would be necessary to replace the synchronous mechanism by a hybrid approach in which synchronization just takes place within fixed-size node sets.

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] N. Fujimoto. Faster matrix-vector multiplication on geforce 8800gtx. In *IPDPS*, pages 1–8. IEEE, 2008.

[2] 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.

[3] Janis Keuper and Franz-Josef Pfreundt. Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability. In *2nd Workshop on Machine Learning in HPC Environments, MLHPC@SC, Salt Lake City, UT, USA, November 14, 2016*, pages 19–26, 2016.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, 2017.

[5] Thorsten Kurth, Jian Zhang, Nadathur Satish, Ioannis Mitliagkas, Evan Racah, Md. Mostofa Ali Patwary, Tareq M. Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail Smorkalov, Jack Deslippe, Mikhail Shiryaev, Srinivas Sridharan, Prabhat, and Pradeep Dubey. Deep learning at 15pf: Supervised and semi-supervised classification for scientific data. *CoRR*, abs/1708.05256, 2017.

[6] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*, pages 451–460, 2010.

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

[8] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[9] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 1–9, 2015.

[10] Ruben 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.

[11] 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.

[12] Yang You, Aydin Buluç, and James Demmel. Scaling deep learning on GPU and knights landing clusters. *CoRR*, abs/1708.02983, 2017.

[13] 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.

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