Continuous Learning of HPC Infrastructure Models using Big Data Analytics and In-Memory processing Tools

Francesco Beneventi[†], Andrea Bartolini[†§], Carlo Cavazzoni[‡] and Luca Benini[†§]
[†]Department of Electrical, Electronic and Information Engineering (DEI), University of Bologna, Italy
{francesco.beneventi, a.bartolini, luca.benini}@unibo.it
[§]Integrated Systems Laboratory, ETH Zurich, Switzerland {barandre, lbenini}@iis.ee.ethz.ch
[‡]Cineca, Italy {c.cavazzoni}@cineca.it

***Abstract*—Exascale computing represents the next leap in the HPC race. Reaching this level of performance is subject to several engineering challenges such as energy consumption, equipment-cooling, reliability and massive parallelism. Model-based optimization is an essential tool in the design process and control of energy efficient, reliable and thermally constrained systems. However, in the Exascale domain, model learning techniques tailored to the specific supercomputer require real measurements and must therefore handle and analyze a massive amount of data coming from the HPC monitoring infrastructure. This becomes rapidly a "big data" scale problem. The common approach where measurements are first stored in large databases and then processed is no more affordable due to the increasingly storage costs and lack of real-time support. Nowadays instead, cloud-based machine learning techniques aim to build on-line models using real-time approaches such as "stream processing" and "in-memory" computing, that avoid storage costs and enable fast-data processing. Moreover, the fast delivery and adaptation of the models to the quick data variations, make the decision stage of the optimization loop more effective and reliable. In this paper we leverage scalable, lightweight and flexible IoT technologies, such as the MQTT protocol, to build a highly scalable HPC monitoring infrastructure able to handle the massive sensor data produced by next-gen HPC components. We then show how state-of-the art tools for big data computing and analysis, such as Apache Spark, can be used to manage the huge amount of data delivered by the monitoring layer and to build adaptive models in real-time using on-line machine learning techniques.**

I. Introduction

It is expected that exascale computing systems will have thousands of nodes each embodying hundreds of cores, large memory hierarchies and featuring a wide range of parallel accelerators, as well as a complex inter-node communication system. Such a complexity and variety of components require large efforts to guarantee efficiency in parallel system programming, application execution and power consumption [1], [6], [13]. Moreover the increased number of components inevitably grows the hardware failure risk putting serious concern about exascale system reliability and MTTF [4]. In this scenario it is clear that the real-time evaluation of the applications performance, together with the accurate measurement of the node energy consumption, are of fundamental importance. Monitoring infrastructure are widely used in the HPC field at any level. The main usage of the monitoring data is oriented to the HPC facility management such as HW diagnostic for failure prevention, resource availability, capacity utilization, cooling systems, energy consumption, billing and application performance [5], [15], [16]. The fast growing demand of performance and data volume, dictated by the impending exascale milestone, require a facility management that can proactively react to failure events, thermal emergencies and unexpected high power consumption phases. Predictive models represent one of the tool that can help in optimal design choices and fast decision making [16]. In the design stage, models help operators to optimally tune and size data-centers components for a given energy and performance budget. Models can be used also during the data-center operational lifetime. When cross coupled with a monitoring infrastructure, they use real-time data to build predictions about energy efficiency, reliability, and availability of the whole system given a certain workload scenario as input [3], [16]. As an example authors in [3], [12] show that power capping techniques can take advantage of workload based power models to efficiently schedule jobs in power constrained systems. However, final performance (energy/cooling efficiency and capacity gain) are bounded by the accuracy of the model itself.

Common techniques to build models are composed of several steps. (i) Data measurement: relevant sensor data are collected and saved in storage system as databases. (ii) Model computation: offline regressive algorithms or machine learning approaches are used to generate the mathematical model. (iii) Model validation: this stage asserts the model quality and usability. From the exascale point of view, the model design flow described above opens several challenges: (i) The huge number of components to be monitored dramatically grows the data to be measured and stored, bringing to high storage costs. (ii) Models learned offline often lack accuracy and suffer non-linear effects when applied to on-line applications and the operating-point is too far from the dataset used in the learning stage. (iii) Validation of the model can be made only taking into account past measured data; future unknown events may generate unwanted and erroneous model outputs, and thus producing wrong predictions.

From the previous analysis it is clear that off-line model learning techniques are not effective in providing fast responses to the extreme huge number of events that can happen in an exascale system. In contrast, on-line model learning techniques can be the answer to fast model adaptability and thus high accuracy. However, new challenges arise as the handling and elaboration in real time of massive quantity of data. "Big data" and "fast data" are hot topics today in the scientific community [7] that is involved in the development of tools capable to ingest and elaborate enormous quantity of real-time data in a streaming fashion [11].

In this paper we adopt Apache Spark, one of the emerging framework for cluster-based computing with stream processing

978-3-9815370-8-6/17/$31.00

and in-memory computing capability, as a tool for the on-line model learning stage. The monitoring infrastructure foundations instead are based on the Message Queuing Telemetry Transport (MQTT) technology, which guarantees scalable and flexible data sharing [8]. Moreover, we present a full machine learning pipeline, oriented to build models for HPC infrastructures, including both the monitoring layer and the learning stage. Our contributions are:

- A scalable and lightweight MQTT-based monitoring infrastructure, which has the task of collecting meaningful sensor data from all the relevant HPC hardware components and software applications. The collector components exploit the MQTT protocol to exchange data with the upper layers of the monitoring framework and have specialized functions to access the multitude of sensors infrastructure present in the HPC nodes, such as in cores PMU, IPMI, GPU and many others.
- A complete Spark back-end architecture as a streaming application able to interfacing to the monitoring layer and build models on-line. The overall infrastructure is completely reusable for the learning of different kinds of models, thanks to the flexibility derived by the MQTT protocol semantic.
- In this work, we specialized the learning algorithm to build the power model of the CPUs in the node as a use case.
- Characterization of the performance of the system both in a single and multi-node setting, with meaningful use cases of online modeling.

The paper is organized as follow. Section II shows the related work. Section III introduces the monitoring framework which will feed the data to the big data analytic engine which is presented in Section IV. Finally Section V quantifies the framework performance in a practical use-case.

II. Related Work

There are today several tools available for monitoring a computing-cluster with different focuses. Ganglia [10] is widely used by system administrators to monitor the system status and resource usage and spot out maintenance problem. For these reasons, it has not been designed for precise sampling time and accurate measurements and thus cannot be used for fine-grain monitoring and application profiling. In addition the internal monitored data redundancy and the polling measurement approach impose trade-offs between scalability and granularity. Profiling libraries [2] provides APIs for accessing architectural events and physical run-time parameters directly from the computing nodes and monitored resources. While this approach is suitable for application profiling, it comes with a intrinsic overhead and an induced-perturbation in the observed system. The msr-safe [14] kernel module in Intel CPUs allows to directly access the performance counter registers enabling customized and low-intrusive monitoring frameworks. However, when targeting monitoring of large-scale systems the communication protocols become a limiting factor. In recent years the MQTT protocol has emerged as a lightweight and scalable data-exchange protocol. It is characterized by a small packet header composed of only two bytes. It implements the publisher-subscriber communication protocol and allows to trade-off latency, overhead and transmission quality by mean of three QoS levels [8]. Several frameworks for real-time data analytics are emerging to address the fast processing of big data [11]. Apache Spark offers, among others, both batch and streaming processing capabilities, several interfaces for different data sources, a full and well supported machine learning library (MLlib) and the support of several programming languages for the application development. To the best of our knowledge this is the first work which combines an MQTT based monitoring framework in HPC computing systems with big data analytics and in-memory processing using Spark to delivery a scalable and flexible real-time model learning tool for large-scale green HPC systems.

III. Monitoring Framework

In this section, we give a high-level description of the monitoring infrastructure. In the next sections we will describe in detail the relevant components used for this paper.

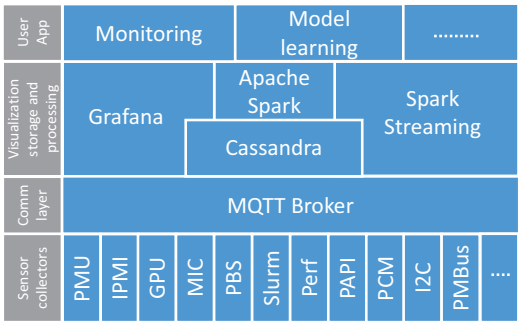

Fig. 1. Model-learning framework

A. System overview

The model-learning framework described in this paper is composed of several components. With the help of the hierarchical view showed in Fig.1 we can distinguish four main groups. Starting from the bottom:

Sensor collectors: These are the low-level components having the task of reading the data from the several sensors scattered across the system and deliver them, in a standardized format, to the upper layer of the stack. These software components are composed of two main objects, the MQTT API and the Sensor API object. The former implements the MQTT protocol functions and it is the same among all the collectors while the latter implements the custom sensor functions related to the data sampling and is unique for each kind of collector. Considering the specific sensor API object, we can distinguish collectors that have direct access to hardware resources like PMU, IPMI, GPU, MIC, I2C and PMBUS and collectors that sample data from others applications as batch schedulers (PBS and Slurm) and tools as perf, PAPI, and PCM.

Communication layer: The framework is built around the MQTT protocol. MQTT implements the publish-subscribe messaging pattern and requires three different agents to work: (i) The publisher, having the role of sending data on a specific topic. (ii) The subscriber, that needs certain data so it subscribes to the appropriate topic. (iii) The broker, that has the functions of (a) receiving data from publishers, (b) making topics available to subscribers, (c) delivering data to subscribers. The basic MQTT communication mechanism is as follows. When a publisher agent sends some data having a certain topic as a protocol parameter, the topic is created and available at the broker. Any subscriber to that topic will receive

the associated data as soon as it is available to the broker. In this scenario, collector agents have the role of publishers.

Visualization, storage and processing: Data published by the collectors is currently used for three main purposes: (i) Real time visualization using web-based tools (Grafana). (ii) Short-term storage in NoSQL databases (Cassandra) useful both for visualization and for batch processing (Apache Spark). (iii) Real time data processing (Spark Streaming). We developed the adapters to interface Grafana and Cassandra to the MQTT broker and thus to the data published by the collectors. The adapters are MQTT subscriber agents that establish a link between the communication layer and every specific tool. Apache Spark instead has its own MQTT receiver that however, as we will see in the following sections, we slightly modified to improve the data management.

User applications: Finally, in the upper layer of the stack, there are all the other applications that can be built on top of the layers below, as infrastructure monitoring, model learning, process control, data analytics and so on.

B. CPU power modeling

The focus of this work is to show how, the general-purpose infrastructure described in section III-A, can be specialized to do on-line model learning using big data processing tools and real time measured data. In this section, we introduce the model that we will use as a test case.

As a proof of concept, we implemented an on-line learning pipeline to determine the parameters of a simple power model of a computing node CPUs. More in detail, the target machine is a classical dual CPU node based on an Intel Haswell processor with $N_c = 16$ total cores (8 per CPU). We use the following equation to formalize the model of the CPU power:

$$\mathrm{P}_{pkg0+1} = \sum_{i=0}^{N_c-1} (a_i + b_i \mathrm{IPS}_i)\mathrm{f}_i \qquad (1)$$

where P_{pkg0+1} is the sum of the CPU0 and CPU1 package power, a_i and b_i are the fitting parameters, while IPS_i and f_i are respectively the instructions per second and the frequency of the i-th core. The dynamic power of the core is mainly modeled by the term $b \cdot \mathrm{IPS} \cdot \mathrm{f}$ where the IPS is used as a proxy to track the core switching activity. As can be seen, this model neglects a constant term that should model the power when $\mathrm{f} = 0$. This condition however is not directly measurable in our case since, during its normal operation, the CPU frequency range is between 1.2GHz and 3.2GHz, so the term $a \cdot \mathrm{f}$ takes into account the core power when it is in idle state.

C. Data measurements

As described in equation (1), the model relies on the per-core frequency, IPS and per-CPU package power consumption. To measure these quantities we used the PMU sensor collector. This component is a per-node daemon process that accesses the PMU of each core and samples, at a given T_{samp} rate, the performance counters using low level MSR read/write operations. It is also capable of accessing the RAPL registers available in the Intel processors to collect the per-CPU power consumption data. The PMU collector, at the end of the sampling stage, delivers each metric to the MQTT broker under a hierarchical topic structure: for the per-core metrics it is `<organization name>/<cluster name>/<node name>/pmu/core/<core number>/<metric name>` while, in the case of per-CPU metrics, it is `<organization name>/<cluster name>/<node name>/pmu/cpu/<cpu number>/<metric name>`. The payload of the MQTT message is in both the cases: `<value>;<timestamp>`. The MQTT broker is a daemon process that can run on non-computing nodes as login nodes to minimize infrastructure intrusiveness. Now that the data is available at the broker, as a temporal sequence of samples, it can be ingested and processed by the computing engine. In the next section we will describe how MQTT data is loaded into Spark and how is preprocessed for the learning stage.

IV. Spark

A. Spark

In this work we use Apache Spark version 1.6.1, and more precisely the Spark Streaming extension, to process in real time the data available on the MQTT broker. Spark Streaming is built on top of Spark (core) so we first introduce its basic concepts. Finally, we explain the basic Spark Streaming building blocks and how it is interfaced with the MQTT monitoring infrastructure.

Spark is a general-purpose analytics computing engine for very large-scale data processing. What is unique in Spark is its ability to share data between processing steps due its in-memory computing architecture. The Resilient Distributed Dataset (RDD) [17] is at the core of the Spark framework. A RDD is a data structure that collects partitions. Each partition contains a subset of the original data loaded into Spark. The key concept is that data is processed serially within a partition and each partition can be processed in parallel. Once the data is loaded into Spark and become an RDD, the data is partitioned implicitly (accordingly to the numbers of cores present in the Spark cluster) or based on user parameters. Thus, the number of parallel workers that execute in a Spark cluster are strictly related to the number of partitions in each RDD. Spark is provided with some fundamental extensions: (i) SparkSQL: This allows Spark to efficiently accesses SQL and NoSQL databases for batch processing. (ii) Spark GraphX: This library enables graph processing on connected entities. (iii) Spark Streaming: This adds the real-time data processing capabilities to Spark. (iv) SparkMLlib: This is a collection of machine learning algorithms tailored on the Spark data processing model.

B. Spark Streaming

To enable real-time processing, Spark Streaming introduces the concept of Discretized Stream (DStream). A DStream is a temporal sequence of RDDs created by dividing the input stream of data in small batches. Thus, the transformations executed on a DStream are executed on every RDD using the Spark (core) engine. Transformations produce other DStreams containing modified RDDs. DStream supports output operations, analogous to the actions for RDDs, that enable the transfer of the processing results to external systems or filesystems.

In Fig.2 is summarized the basic Spark Streaming architecture coupled with the MQTT monitoring infrastructure. The Spark user program executes on the *driver* process. It sends *tasks* to *executors* that run on the *worker* nodes distributed along the cluster. The *input receiver* is a special task that connects with external data sources to receive streaming data. The Spark framework provides, among others, a basic *MQTT receiver* that subscribes to topics and returns messages. However the current version of this component implements a very

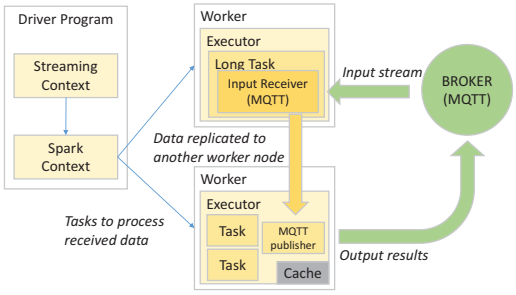

Fig. 2. Spark Streaming architecture and the MQTT interface

basic subset of the MQTT functionality. We modified the original MQTT receiver adding some features such as the "topic name" (associated to the received message) in the returned data structure. We will see later how this information is essential for the data flow management into Spark. The receiver divides the input stream in blocks forming the sequence of RDDs. At every batch interval the driver instantiates tasks on the workers to process these blocks of data. Finally, the results are sent out from Spark. We implemented a *MQTT publisher* object that is created only once on the worker nodes and is reusable among different tasks. The publisher takes the results of the processing stage and sends them back to the MQTT broker, making data available to other external components, such as data visualization systems or databases. In the next section, we will describe the model learning application that is built on top of the software architecture described above.

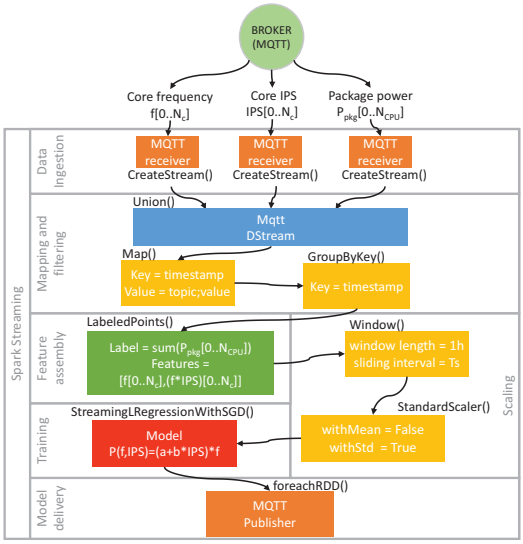

Fig. 3. Spark Streaming driver program for model learning

C. Model learning application

In this section, we describe the main steps involved in the model training using the Spark Streaming API. We also show how the hierarchical structure of the topic namespace can be used to easily manage the Spark data-flow and build scalable algorithms. To train the power model (1) we used SparkMLlib, the scalable machine learning library built around the Spark framework. This ensures high performances since algorithms well exploits the in-memory computing capabilities of Spark. For the training of on-line models the SparkMLlib library provides the StreamingLinearRegressionWithSGD() API. It essentially implements a streaming linear regression using the Stochastic Gradient Descent (SGD) method. This algorithm solves the following optimization problem:

$$\min_{\mathbf{w}\in\mathbb{R}^d} f(\mathbf{w}) \tag{2}$$

where $\mathbf{w}$ is the vector of weights having d components and

$$f(\mathbf{w}) := \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{w};\mathbf{x_i},y_i) \tag{3}$$

where $\mathbf{x_i}\in\mathbb{R}^d$ is the training vector or features and $y_i\in\mathbb{R}^d$ is the data we want to predict or labels. The loss function is:

$$L(\mathbf{w};\mathbf{x_i},y_i) := \frac{1}{2}\left(\mathbf{w^T x}-y\right)^2 \tag{4}$$

In the streaming regression algorithm, at each sample of the input stream, the optimization algorithm executes t iterations to update the weights $\mathbf{w}$ as: $\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} - \gamma f'_{\mathbf{w,i}}$ where γ is the step-size or learning rate hyper-parameter and $f'_{\mathbf{w,i}}$ is the stochastic subgradient.

We need to map our power model (1) to the optimization problem described above in order to use the SparkMLlib API. The weights vector is formed as $\mathbf{w} = [a_0,\ldots,a_{N_c},b_0,\ldots,b_{N_c}]$ while, assuming to have $\mathbf{f} = [\mathrm{f}_0,\ldots,\mathrm{f}_{N_c}]$ as the vector of the per-core frequencies and $\mathbf{IPS} = [\mathrm{IPS}_0,\ldots,\mathrm{IPS}_{N_c}]$ as the vector of the per-core IPS, the final features vector is:

$$\mathbf{x} = [\mathrm{f}_0,\ldots,\mathrm{f}_{N_c},(\mathrm{f}\cdot\mathrm{IPS})_0,\ldots,(\mathrm{f}\cdot\mathrm{IPS})_{N_c}] \tag{5}$$

and the corresponding labels are:

$$y = \sum_{i=0}^{N_{CPU}-1} \mathrm{P}_{pkg_i} \tag{6}$$

The MLlib API provides the LabeledPoints object, composed by the labels and the features vectors. It is used as input dataset to the learning algorithms implemented in the library. One of the main tasks of the Spark driver program is to transform the stream of data, arriving from the MQTT broker, into an ordered sequence of LabeledPoints objects that are going to feed the learning algorithm. The Spark program used for the model learning is described in Fig. 3 and is composed by the following stages:

Data ingestion: In this first step, the program creates the MQTT receivers that subscribe to the selected topics and receive messages. In this stage, we leverage the wildcards provided by the MQTT protocol to easily subscribe to the per-core metrics. For example, assuming the topic hierarchy defined in Section III-C, using dummy names as example, we can subscribe to the frequency topics of all the cores of the node named "node001" using `"AAA/BBB/node001/pmu/core/+/freq"` as a topic in the MQTT receiver. Doing the same for the remaining metrics, the modified version of the MQTT receiver returns a DStream containing objects formed as `<topic name>;<message>`. The `<topic name>` field is critical because contains the metric name and the core number information needed to correctly build the feature vector. The `<message>` field contains the MQTT payload that, in our system is defined as `<value;timestamp>` corresponding to the metric.

Mapping and filtering: In this stage, we ensure that the messages are temporally consistent. Using a sequence of transformations, we create a unique DStream of key-value objects grouped and sorted by key (timestamp).

Feature assembly: In this step we build the labels and features vectors (LabeledPoints) as defined in (5). To properly build the vectors, we use the core index information and the feature names stored in the topic name fields of the object.

Scaling: We standardize the features before passing the data to the learning algorithm. Due to the large variance of the measured values, especially for the IPS feature, the algorithm performance can decrease. We scaled all the feature to unit variance before the training. The features variance is calculated on a time window of one hour of data at each interval of the streaming algorithm.

Training: Finally the model is trained using the algorithm described in this section.

Model delivery: At each streaming batch interval, the resulting model parameters are published to the MQTT broker. Any external system that needs an updated model can subscribe to the corresponding topic and receive the parameters in a timely fashion.

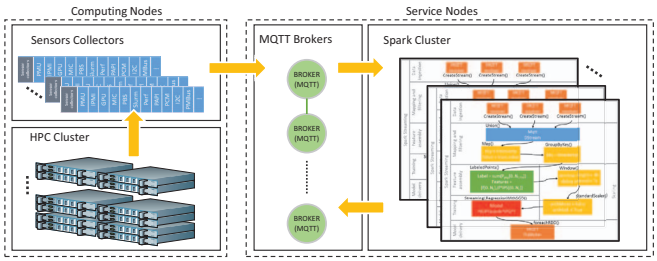

Fig. 4. Scalable model learning infrastructure

V. Results

In this section we analyze the accuracy, scalability and overhead of the proposed modeling framework. We first created a test environment following the architecture showed in Fig. 4 and using a mini-cluster composed by 8 nodes of a working production system (Galileo at Cineca) as a case study. Each node is equipped with two 8-cores Intel Haswell CPUs (E5-2630 v3 @ 2.40GHz) and 128Gb of DRAM. We instrumented the mini-cluster with the MQTT monitoring infrastructure installing on each node the "PMU collector" while the "Broker" and the "Spark cluster" were installed on a service node (Intel Haswell E5-2670 v3 @ 2.30GHz, 24 cores and 128Gb DRAM).

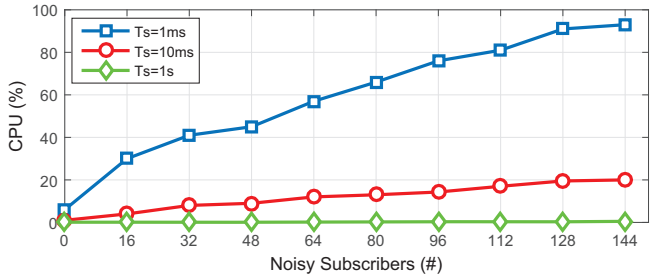

Fig. 5. Broker process overhead

Broker overhead: We used Mosquitto [9] as a MQTT broker. It is a single thread user space process and we measured its overhead (CPU usage) in different scenarios. In Fig. 5 we considered a progressive number of concurrent clients (Noisy Subscribers) that publish messages at different sampling rates. It results that its overhead, for sampling rates greater than 1 second, is negligible and has a low sensitivity to the increase of the clients number. Considering these results, the broker process can be executed also on the target node when low latency in the MQTT packets is required. We measured the latency considering two cases: broker running on the frontend and on the computing node. Figure 6 shows the

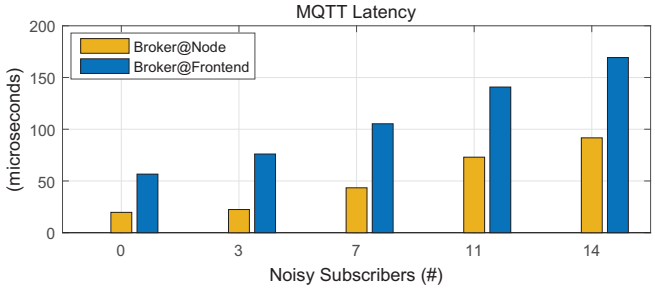

Fig. 6. MQTT packets latencies

delay of a MQTT packet (40 byte) delivered by the broker. We considered the worst cases (on the x-axis) of multiple "noisy subscribers" that concurrently publish messages on the broker to increase its overhead and to measure their influence on the packets latency. As expected, the latency is more than halved when the broker is executed on the node but, in this case, the broker workload should be taken into account. Though Mosquitto is a single thread process, it offers the opportunity to link more broker processes together (bridging), sharing the same topics namespace and thus allowing a scalable broker infrastructure.

PMU collector overhead: The overhead of this component is critical since it executes on the cores of the target nodes and should have the lowest possible intrusiveness in the system. It is a single thread process. We measured its CPU usage at different sampling intervals, obtaining the results showed in Fig. 7. For instance, considering a sampling rate of 20ms, the overhead is less than 10% of the usage of a single core, while for the case of 1 second it decreases to less than 1%.

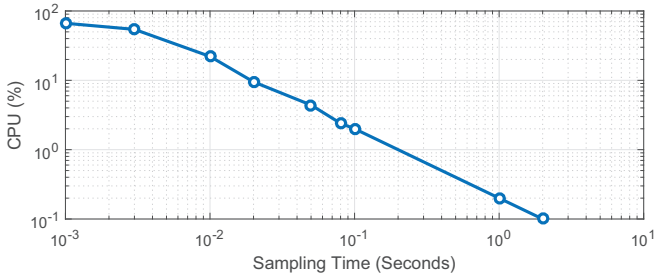

Fig. 7. PMU collector overhead

Model accuracy: We evaluated the accuracy of the model learning application described in Section IV-C. In this test, we executed eight learning applications (Spark jobs) one for each node of the target mini-cluster, in order to train one power

model per node in real-time. The submitted programs are exactly the same for each job, except for the "node name" field in the MQTT topic. The sampling rate of the PMU collectors is $T_{samp} = 2$ seconds. In Fig. 8a is showed the evolution

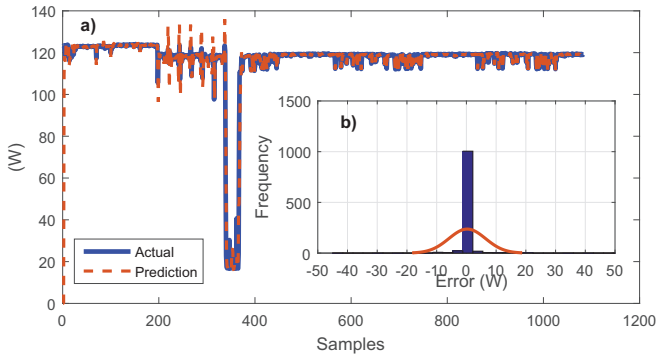

Fig. 8. Measured power vs. model prediction

of the model output compared to the real measurements. The Figure highlights the transition between two different workload phases running on the node. The on-line training algorithm is capable to capture in real-time these behaviors and quickly adapts the model parameters as showed in Fig. 9. The accuracy of the model is depicted in Fig. 8b denoting in this case an average error of 0.0254W.

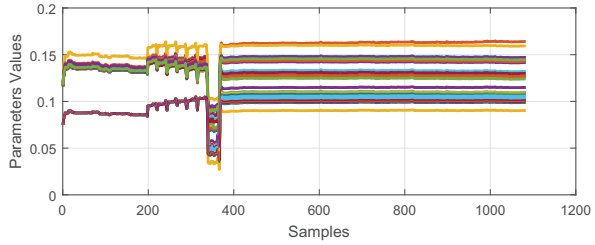

Fig. 9. Power model parameters

Spark overhead: Finally we evaluated the resource needed by the Spark jobs in terms of CPU and DRAM usage of the service node. At every sampling interval T_{samp}, new data arrive in the Spark cluster and, as a consequence, a new set of tasks is scheduled and executed by the computing engine. In

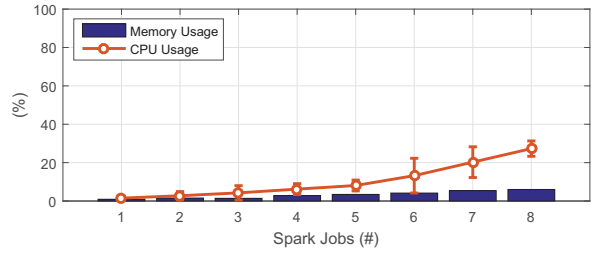

Fig. 10. Resources used by the model learning applications

Fig.10 we considered the cases with an incremental number of learning applications concurrently running (x-axis). The error bars in the plot show the minimum and the maximum CPU usage corresponding to the idle and computing stages, while the point in the middle represents their average value.

VI. Conclusion

In this paper, we introduced a scalable model-learning framework tailored for the upcoming exascale HPC systems. We presented the main building blocks, starting from the monitoring infrastructure, based on scalable and lightweights communication protocols, until the application layer. As a use case, we showed how is possible to train on-line models using Apache Spark, a real-time distributed in-memory analytics engine, that processes data coming from the HPC equipment. The application implemented in this paper can be largely reused to train a broad range of complex models, replacing the basic model used in this work as a test case. Moreover, we showed how the flexibility of the MQTT protocol alleviates the complexity of the data flow management, augmenting the overall modelling capabilities. Finally, we evaluated the accuracy, overhead and scalability issues of the main framework components, such as the sensor data collector, the message broker and the stream processing engine.

VII. Acknowledgments

This work was supported, in parts, by the FP7 ERC Advance project MULTITHERMAN (g.a. 291125), by the EU H2020 FETHPC project ANTAREX (g.a. 67623) and by the EU H2020 FETHPC project Exanode (g.a. 671578).

References

[1] S. Ashby et al. The Opportunities and Challenges of Exascale Computing: Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee. Technical report, (ASCAC), 2010.

[2] S. Benedict. Energy-aware performance analysis methodologies for HPC architectures: An exploratory study. *Journal of Network and Computer Applications*, 35(6):1709 – 1719, 2012.

[3] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini. *Predictive Modeling for Job Power Consumption in HPC Systems*, pages 181–199. Springer International Publishing, Cham, 2016.

[4] F. Cappello. Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities. *Int. J. High Perform. Comput. Appl.*, 23(3):212–226, Aug. 2009.

[5] C. Conficoni, A. Bartolini, A. Tilli, G. Tecchiolli, and L. Benini. Energy-aware cooling for hot-water cooled supercomputers. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1353–1358. EDA Consortium, 2015.

[6] J. Dongarra et al. The International Exascale Software Project roadmap. *Int. J. High Perform. Comput. Appl.*, Jan. 2011.

[7] C. K. Emani, N. Cullot, and C. Nicolle. Understandable big data: A survey. *Computer science review*, 17:70–81, 2015.

[8] S. Lee, H. Kim, D. k. Hong, and H. Ju. Correlation analysis of mqtt loss and delay according to qos level. In *The International Conference on Information Networking 2013 (ICOIN)*, pages 714–717, Jan 2013.

[9] R. Light. Mosquitto-an open source mqtt v3. 1 broker, 2013.

[10] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[11] S. J. Morshed, J. Rana, and M. Milrad. Open source initiatives and frameworks addressing distributed real-time data analytics. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1481–1484, May 2016.

[12] O. Sarood et al. Maximizing throughput of overprovisioned HPC data centers under a strict power budget. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 807–818. IEEE Press, 2014.

[13] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *International Conference on High Performance Computing for Computational Science*, pages 1–25. Springer, 2010.

[14] K. Shoga, B. Rountree, M. Schulz, and J. Shafer. Whitelisting msrs with msr-safe. In *3rd Workshop on Exascale Systems Programming Tools, in conjunction with SC14*, 2014.

[15] H. Shoukourian et al. Monitoring power data: A first step towards a unified energy efficiency evaluation toolset for HPC data centers. *Environmental Modelling & Software*, 56:13 – 26, 2014.

[16] A. Srbu and O. Babaoglu. Towards data-driven autonomics in data centers. In *Cloud and Autonomic Computing (ICCAC), 2015 International Conference on*, pages 45–56, Sept 2015.

[17] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.