Spark Deployment and Performance Evaluation on the MareNostrum Supercomputer

Ruben Tous*†, Anastasios Gounaris‡, Carlos Tripiana*, Jordi Torres*†,
Sergi Girona*, Eduard Ayguadé*†, Jesús Labarta*†, Yolanda Becerra*†, David Carrera *† and Mateo Valero*†
*Barcelona Supercomputing Center (BSC). Barcelona, Spain
†Universitat Politècnica de Catalunya (UPC). Barcelona, Spain
‡Aristotle University of Thessaloniki, Thessaloniki, Greece
Email: {ruben.tous; carlos.tripiana; jordi.torres; sergi.girona; eduard.ayguade; jesus.labarta; yolanda.becerra; david.carrera; mateo.valero}@bsc.es, gounaria@csd.auth.gr

Abstract—**In this paper we present a framework to enable data-intensive Spark workloads on MareNostrum, a petascale supercomputer designed mainly for compute-intensive applications. As far as we know, this is the first attempt to investigate optimized deployment configurations of Spark on a petascale HPC setup. We detail the design of the framework and present some benchmark data to provide insights into 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 Big Data workloads on a computing system that is based on the compute-centric paradigm. Further, we derive conclusions aiming to pave the way towards systematic and optimized methodologies for fine-tuning data-intensive application on large clusters emphasizing on parallelism configurations.**

I. Introduction

The work described in this paper explores the performance and scalability of Apache Spark [1], deployed on a real-world, petascale, HPC setup, the MareNostrum supercomputer.[1] Spark may be deemed as an evolution of Hadoop [2], aiming to benefit from memory availability, elegantly handling iterations and being suitable for both batch and streaming jobs; overall it is shown to outperform Hadoop for many applications by orders of magnitude [3], [4]. Our framework is named Spark for MareNostrum (or *Spark4MN*) and allows to efficiently run a Spark cluster over a Load Sharing Facility(LSF)-based environment while accounting for the hardware particularities of MareNostrum.

Apart from deployment, the next biggest challenge in massively parallel big data applications is scalability and proper configuration. Simply running on hundreds or thousands of cores may yield poor benefits (e.g., as in [5]) or even degraded performance due to overheads [6]. We deal with this issue and we aim to make the first step towards systematic analysis of the several parameters and optimized configuration. Spark4MN provides functionalities to evaluate different configurations in terms of cores per execution managers, networking, CPU affinities, and so on. This has allowed us to perform multiple optimizations and to identify solutions to potential inefficiencies of a data-centric framework running over a compute-centric infrastructure. More specifically, we evaluate the behavior of two representative data-intensive applications, sorting and k-means. 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 data-intensive applications. Overall, our conclusions, as presented hereby, aim to assist both HPC administrators to deploy Spark and Spark developers having access to HPC clusters to improve the performance of their applications.

In the next section, we provide background information about Spark. Sec. III discusses related work.The Spark4MN job submission framework is presented in Sec. IV. Then, we discuss the benchmarking applications (Sec. V) and the experimental results, aiming to shed light onto the parameters that have the biggest impact and their effective configuration (Sec. VI). We conclude in Sec. VII.

II. 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 [3]. 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

[1]This work is partially supported by the Spanish Ministry of Economy and Competitivity under contract TIN2012-34557, by the BSC-CNS Severo Ochoa program (SEV-2011-00067), and by the SGR programme (2014-SGR-1051) of the Catalan Government.

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., $O(m^2)$.

III. RELATED WORK

In [5], a framework is described to enable Hadoop workloads on a Cray X-series supercomputer. The authors evaluate the framework through the Intel HiBench Hadoop benchmark suite and a variety of storage backends, including node local memory (RAM) storage, node local hard drive storage, and a shared Lustre filesystem used as either a direct file store or with HDFS layered over it. The results show that, in most cases, local storage performed slightly better than Lustre.The memory-resident implementation of Spark is a key difference with Hadoop, and thus the results of this work are not directly comparable with ours. In [7], 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. Our investigation is orthogonal, since we focus on scalability and configuration properties.

In [8], 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. They also propose a set of Big Data applications as a benchmark to investigate and evaluate the different paradigms. This work shows experimental results for the k-means algorithm running on Hadoop, Mahout, MPI, Python-based Pilot-k-means, HARP and Spark. In our work, we also employ k-means as a benchmark application, but we complement it with another one, for which the effects of data shuffling are even more prominent, and overall we perform a different kind of experiments. In addition, there are several efforts that have investigated the deployment of HPC applications on cloud-based data centers, e.g., [9], [10].

Efficient configuration of MapReduce environments is a topic that has attracted a lot of interest recently. Some proposals aim to optimally configure Hadoop through capitalizing on previous executions logs and using a what-if engine to predict the behavior of the system under different configurations (e.g., [11]). Other techniques employ task profiling and optimization algorithms for searching the space of all possible configurations (e.g., [12]). A large portion of efforts have been devoted to addressing the problem of skew and load balancing among workers (e.g., [13]) and mitigating the impact of data shuffling (e.g., [14]). Nevertheless, the main research topic has been the development and re-engineering of data management algorithms for the MapReduce setting.There is no work to date that aims to investigate the optimal configuration of Spark applications on large clusters, apart from a few recommended best practices' tips in books, such as [15] and websites[2]. Our work aims to scratch the surface of this important topic and initiate the systematic research towards understanding how to best utilize such an emerging dataflow framework on HPC infrastructures.

IV. THE SPARK4MN FRAMEWORK

We have designed and developed a framework (Spark4MN) to efficiently run a Spark cluster on MareNostrum. Spark4MN runs over IBM LSF Platform workload manager, but it can be ported to clusters with different managers (e.g., Slurm manager) and does not rely on Spark cluster managers, such as Apache Mesos and Hadoop YARN. 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*.

Marenostrum Overview: 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

[2]http://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices

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.

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

V. BENCHMARKING APPLICATIONS

We have selected two representative benchmarking applications, namely *sort-by-key* and *k-means*, which are also part of *HiBench*[3]. We have run these applications over synthetic data generated with the *spark-perf* Spark performance testing framework [4] to allow for easy reproduction of our experiments by others. The selected applications are characterized by different challenges in terms of their efficient parallelization. *sort-by-key*'s performance is largely bounded by the communication cost during shuffling, whereas *k-means* requires a significant amount of resources in terms of both CPU and networking. These challenges are representative of two generic types of real applications: a) one where (iterative) applications can be parallelized in a straightforward manner with relatively small shuffling/communication cost between iterations, like k-means; b) another one where the data shuffling cost is more prominent, with sorting being the most representative example of this category since it repartitions the complete dataset, while performing a meaningful job. There are many different parameters that can be configured in both the Spark framework (e.g., number of nodes, cores per node, available memory, affinity, network and storage, and so on) and within the selected benchmarks (e.g., input data size, number of partitions, number of iterations, number of centers, and so on), which renders the efficient parallelization even more challenging.

[3]https://github.com/intel-hadoop/HiBench/

[4]https://github.com/databricks/spark-perf/

k-means is a well-known clustering algorithm for knowledge discovery and data mining. Spark's MLlib includes a parallelized variant of the k-means++ method called k-means$||$ [16]. Overall, *k-means* is both a compute and communication intensive application. When applied to a d-dimensional dataset of n records, the computation complexity is $O(ndk)$, assuming bounded number of iterations. The data transmitted over the network is not proportional to the input size but to the k and d values and the number of data partitions. In each iteration, each task reports the local statistics per intermediate center using a *reduceByKey* transformation. The master collects these intermediate results, computes the global intermediate centers, which are then broadcasted to all tasks for the next iteration. As such, there is a clear trade-off between a lower degree of parallelism and increased communication cost.In our experiments, the intention has not been to modify the MLlib k-means$||$, but to identify the best performing parallelization parameters, when this algorithm runs on top of MareNostrum. On the other hand, *sort-by-key* is a critical operation used by many applications that helps in better revealing the pattern of *shuffle* operations. More formally, the amount of data transmitted over the network is $O(n)$, where n is the amount of input data records to be sorted.

VI. RESULTS

We have submitted and tested several thousands 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. In most cases, we do not exploit the full computational power of MareNostrum but use a more limited amount of cores, motivated by the fact that most Hadoop clusters to date are relatively small [17].

In all experiments, we allocate enough memory resources so that RDDs can fit into the main memory.This, however, does not mean that we do not utilize local and network storage; access to local storage is inevitable due to the way shuffling is handled, whereas for network storage, utilization is performed for test purposes and involves evaluating the overhead of reading source data. In this work, we mostly focus on parallelism configurations, which are identified as particularly significant in determining performance; thorough investigation of storage alternatives, especially when RDDs cannot fit in the main memory, are out of scope and left for future work.

A. Performance and Scalability

The main goal of these experiments is to evaluate the speed-up, scale-up, and size-up properties of the selected algorithms. To this end, we use datasets up to hundreds of GBs of raw data. The size of RDDs is reported to be 2-5 times larger than that; in our experiments 400GBs of data in the sort-by-key application correspond to an RDD of 1TB. The cluster sizes range from 128 cores (i.e., 8 16-core machines) up to 4096 (i.e., 256 machines).

1) K-means speed-up: In the first set of experiments, we keep the input dataset constant and we increase the size of nodes/cores running the Spark application; whenever we refer to nodes, we mean MareNostrum machines that run the executors, while the driver always runs on a separate machine; each machine is equipped with 16 cores and 32 GB of RAM. The main set-up of k-means is motivated by the experiments in the Spark RDD and Shark papers [3], [4], and we set the datasize to 100GBs. We distinguish between 3 cases: (i) 10M vectors of 1000 dimensions (10M1000d); (ii) 100M vectors of 100 dimensions (100M100d); and (iii) 1B vectors of 10 dimensions (1000M10d). In each case, the vectors were randomly generated in memory before execution, k is set to 100, and we allow k-means to run for exactly 10 iterations so that all time metrics refer to the same amount of computation. The algorithmic complexity in all cases is the same, but the three datasets behave differently. The results from 128 (8 nodes) up to 512 cores (32 nodes) are shown in Figure 1, where we can see that for large datasets in terms of number of records, k-means can scale well. In the figure, we present the performance for the most efficient configurations; we discuss these configurations in detail later.

There are three significant observations that can be drawn from Figure 1. First, although the raw data are of the same size containing 10 billion atomic elements thus being of equal algorithmic complexity, their processing cost differs significantly. Second, when there are 1000 dimensions, the speed-up is smaller. This is attributed to the fact that the extra overhead in data shuffling (same number of centroids to shuffle but bigger) partially outweighs the benefits of parallelism. Third, in practice, it is more expensive to process larger datasets with smaller records, as shown by the running times for the 1000M10d dataset. In those cases, even the RDDs as Java objects are larger in size (by more

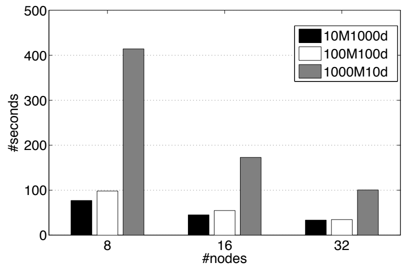
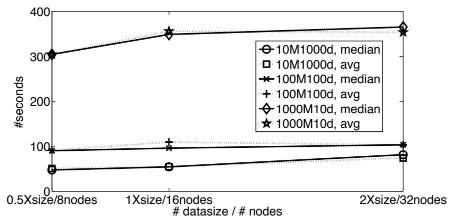


Figure 1: Times for running k-means for 10 iterations.



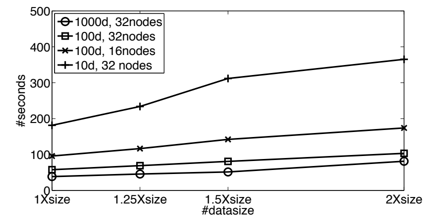


Figure 2: Times for running k-means for 20 iterations showing the scale-up (top) and size-up capability (bottom).

than 20%), and the system is stressed more with regards to memory during shuffling. The latter problem is alleviated when we increase the aggregate memory through increasing the number of nodes, and it is of no surprise that we can observe super-linear speed-up when going from 8 to 16 nodes. Note that similar super-linear speed-ups have been reported for Hadoop applications as well [6].

Finally, in [3], it is shown that the average time per iteration when 400 cores are employed, where each core is equipped with 4GB of memory, is approximately 33secs, without considering the first iteration, which is significantly more expensive. Our results are not directly comparable, as we use up to 512 cores with 2GB of memory, however, the average time per iteration can be as low as 10 secs approximately, without omitting the first iteration. When omitting the first iteration, for 32 nodes and the 100M100d dataset, the average time per iteration drops to 2.2 secs, which is lower than the number of 4.1 secs, recently reported by Databricks.[5]

[5]R. Xin's presentation "Performance Optimization Case Study: Shattering Hadoop's Sort Record with Spark and Scala" during Spark Summit East 2015.

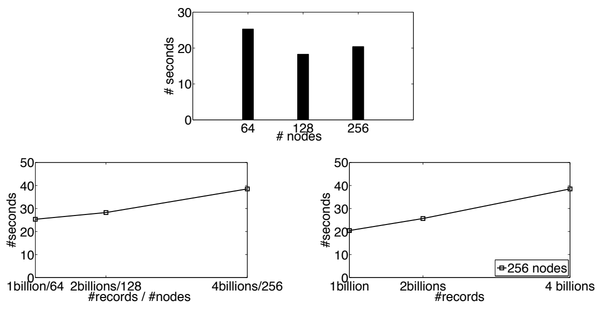


Figure 3: Speed-up (top), scale-up (bottom-left) and size-up (bottom-right) capability for sort-by-key.

2) K-means scale-up: We process the same datasets, and we now modify both the number of records and the number of machines, i.e., the infrastructure scales-out. Apart from the sizes reported above, we also test half and double values, i.e., from 50 to 200GBs of raw data. Further, we increase the number of k-means iterations to 20, in order to decrease the standard deviation per iteration. The results are shown in Figure 2(top). In this figure, we show both the average and the median values. Ideally, all the plots should be horizontal; our system behaves closely to that. The most linear behavior is exhibited by the 100-dimensional dataset, where the largest difference between the three settings is no more than 15%. In line with the results of Figure 1, the 1000-dimensional data set, although runs faster, is the hardest to scale-up linearly, due to the significant increase in the communication cost as explained earlier.

3) K-means size-up: We perform a third set of experiments, to assess the capability of sizing-up. We keep the number of nodes constant (either 16 or 32), and we gradually increase the dataset from 100GBs to 200GBs (raw data sizes). As shown in Figure 2(bottom), Spark4MN exhibits a behavior where the curves are (almost) linear. Processing 100M 100-dimensional records takes 57.8 secs (resp. 95.8 secs) when using 32 (resp. 16 machines); the elapsed time to run 20 iterations of k-means on a 200M dataset is 103.2 secs (resp. 174 secs). Moreover, when we process the 10-dimensional dataset and we double the size, the processing time grows from 181.3 to 365.1 secs, i.e., just over 2 times.

4) Sort-by-key speed-up, scale-up and size-up: In this experiment, we sort string key-value pairs of 100 bytes each (key: 10 bytes, value: 90 bytes). The amount of data to be shuffled is equal to the amount of source data. This means that there is a significant disk contention during shuffling that stresses the commodity local hard disks. The contention is aggravated by the fact that each local disk is shared among 16 tasks. Moreover, in this experiment we significantly increase the maximum number of cores employed to identify the turning point, where the overheads due to shuffling and parallelism outweigh benefits.

Figure 3 shows the main results. The top figure refers to sorting 100GB of raw data with degree of parallelism from 1024 up to 4096. We can see that, when going to 2048 cores, there is an 1.38X speed-up. However, further increasing the degree of parallelism to 4096, causes a slow down of more than 10%. Of course, this is also due to the relatively small amount of data processed; sorting 200GB of raw data with 4096 cores leads to speed-ups of 10% compared to the 2048 case. In Figure 3(bottom-left) we see that the scale-up curve is not horizontal, thus clearly depicting the effects of data shuffling overhead. Nevertheless, the size-up curve is almost linear, as shown in Figure 3(bottom-right).

How do these figures compare to the 100TB sort behavior reported by Databricks? This is hard to answer, but we provide some indicative numbers. We consider only the cases with 4096 cores, where the shuffling overhead is more prominent so that the numbers are more representative (instead of presenting the highest throughput observed with less cores). The average sorting rate per core in the Databricks case is 0.011 GBs/sec. In our case, it is 0.003GBs/sec. We use similar types of processors. Databricks scientists use 6592 cores, whereas the maximum number of cores we use is 4096 as already mentioned. The communication overhead is proportional to the number of core pairs [6], and we have less than 40% of core pairs than Databricks. However, in this shuffling-intensive scenario, memory and disk is of crucial importance, and our nodes are at least 4 times inferior in this regard. Databricks cores have approx. 8 GBs of memory (compared to 2GBs in our case) and each SSD disk is shared among 4 cores (compared to a single SATA disk shared among 16 cores in our case). Given all these, having approximately 4 times lower average throughput per core is another indication of the efficient deployment of Spark on top of MareNostrum.

B. Parallelism Configuration for k-means

In the previous experiments, we showed the scalability of the applications when run on our infrastructure. Here, we examine the configurations that directly impact on parallelization. More specifically, we assess the impact of (i) the degree of parallelism, (ii) the executor size, and (iii) the processor affinity.

Regarding the degree of parallelism, in our benchmarks (in both k-means and sort-by-key setup) the number of *tasks* is equal to the number of RDD *partitions*. Given that we can change the partitioning (increasing or decreasing the number of partitions), we can experiment with varying number of tasks over the same amount of cores (we keep the default Spark configuration[6] to allocate each task to a single core, i.e., a single task not to be processed by multiple cores). We experiment with cases that overall assign 1, 2 or more partitions per core.

[6]http://spark.apache.org/docs/1.4.0/configuration.html

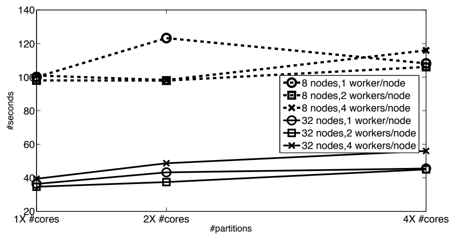


Figure 4: Median times for running k-means for 10 iterations with different number of partitions.

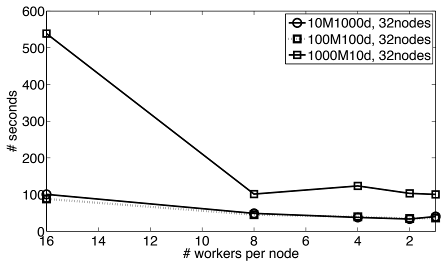


Figure 5: Median times for running k-means for 10 iterations under different worker configurations.

1 16-core worker per node		4 4-core worker per node	
NODE	0.9894	NODE	1.0366
SOCKET	1.7165	SOCKET	1.0541
CORE	11.532	CORE	3.2247

Table I: Impact of core affinities.

The results are that, for k-means, the best-performing configuration is to have as many partitions as cores (contrary to the suggestion in http://spark.apache.org/docs/1.4.0/tuning.html). In our experiments, the performance degradation of setting the number of tasks to 2 times the number of cores can reach up to 31.59%, whereas we have also observed some cases where doubling the number of tasks incurred small improvements (of up to 3%). Two representative examples are shown in Figure 4, where we process the 100M100d dataset with 8 and 32 nodes. The plots show how the performance degrades when using 2 or 4 times more partitions than the number of available cores for three different configurations regarding the allocation of workers per node. When using 8 nodes and doubling the number of partitions, very small improvements have been observed (when using 256 instead of 126 partitions). However, in those examples, using 4 tasks per node can degrade performance up to 42% for 8 nodes and up to 14.82% for 32 nodes (both configured to run 4-core workers). The partition size did not seem to have an impact on our experiments, i.e., the number of partitions could not be derived by simply looking at the size of RDDs.

Another important observation that can be drawn form Figure 4 is that the number of spark workers for a given number of cores is a configuration parameter, which requires special attention. We performed experiments with different distribution of the workers within each node (Spark4MN allows for workers from 1 to 16 cores); for example, 2 exclusive cores per worker, 4 exclusive cores per worker, shared cores, and so on.The larger the worker, the more tasks it has to supervise and control. We experiment with workers from 1 to 16 cores and the results show that, on average, larger workers are preferable. 4 workers per node with 4 cores each can perform up to 30% worse than 2 workers with 8 cores each, or 1 worker per node with 16 cores. Figure 5 shows how the performance degrades when the 16 cores of each node are divided into 1 to 16 workers. When processing the 100M100d with 32 nodes, we can observe that the 2 8-core workers configuration outperforms the setting with 16 single-core ones by a factor of 2.5, i.e, the same amount of cores can yield 2.5 times higher running times. For the 10-dimensional dataset, the performance degradation due to smaller executors is even higher.

Regarding the core affinity, allowing for overlapping degrades performance significantly since k-means is cpu0intensive. We use the 100-dimensional dataset, and we modify the core affinities, testing the different options mentioned in Section IV. Table I presents the normalized values for 32 nodes. 1 corresponds to not setting any affinity, thus allowing the Spark4MN/MareNostrum scheduler to make decisions on resource allocation. We can observe that forcing all cores of a worker to be on the same physical machine may yield negligible benefits (very close to statistical errors). However, when forcing joint use of physical cores, then the performance starts degrading. For example, pinning 16 requested cores to a single socket implies that the corresponding tasks will run on 8 physical cores, and leads to performance degradation by a factor of 1.71. If all 16 tasks are pinned to a single core, then the degradation is of an order of magnitude. For settings, where the executors are smaller, the negative effects of joint use of physical cores are mitigated. Pinning all cores of an executor to a single socket (containing 8 cores) does not lead to significant performance penalty. However, trying to run 4 tasks on a single core, increases the running time by a factor of 3.25. Finally, Spark4MN gives the opportunity to request nodes that are as physically close as possible to minimize the number of switches in-between. In the 100-dimensional dataset, this has not been observed to yield any benefits; on the contrary, due to these restrictions, there were significant delays in scheduling the job that reached up to 52 hours.

C. Parallelism Configuration for sort-by-key

We evaluate the impact of the degree of parallelism and the executor size for sort-by-key. More specifically, we sort 1 billion records using 64 nodes. Figure 6(left) depicts how the performance is affected when allocating 1, 2 or

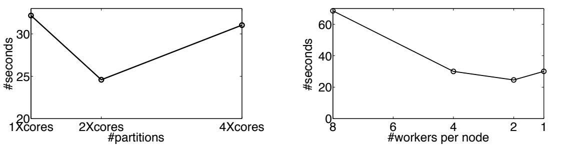


Figure 6: Impact of partition (left) and executor size (right) for sort-by-key.

k-means (32 nodes)		
Dataset	Infiniband	Ethernet
10M1000d	33.455secs	43.164secs
100M100d	34.649secs	35.879secs
1000M10d	103.282secs	108.303secs

sort-by-key (64 nodes)		
Partitions	Infiniband	Ethernet
1024	31.824secs	40.622secs
2048	23.821secs	25.213secs

Table II: Impact of network.

4 partitions per core. Contrary to the previous case, we observe speed-ups when there are 2 partitions per core, but again, significant degradation for 4 partitions per core. This behavior is attributed to the extremely shuffling-intensive nature of the job. In other words, the extent to which shuffling dominates plays a significant role in the optimal number of tasks per core. Figure 6(right) is in line with Figure 5. Having 2 8-core executors instead of 8 2-core ones, improves on the running time by a factor of 2.79 leaving all the other parameters the same. This shows that the worker size depends on the infrastructure rather than on the application.

D. Impact of Network Interface and Storage Architecture

We conduct further experiments that (i) use the Ethernet network instead of Infiniband and (ii) read the initial datasets from DFS instead of generating in and reading from main memory directly. For assessing the impact of the network interface, we fixed the setting of k-means to 32 nodes and 2 8-core workers per node. Table II contains the results. We can observe significant differences only for the 10M1000d dataset, which incurs the highest communication cost due to increased data shuffling. In this case, the performance benefits of Infiniband amount to 22.5% compared to Ethernet. In the other two cases, the benefits are limited to 3.5% and 4.6%, respectively. Sort-by-key is also affected by the network interface. When using a small number of partitions, the relative difference is more than 27%, despite the fact that the major bottleneck due to shuffling is storing on the local disk rather than transmitting data over the network. This difference gets significantly smaller as we increase the number of partitions.

Regarding GPFS, we repeated the k-means experiment with the 100-dimensional dataset (for 32 nodes). When reading from GPFS directly, the median running time exceeds 200secs (5.47X increase). When reading from GPFS and caching, the running time increase dropped to the half. Reading from GPFS mainly impacts the first iteration of the k-means algorithm. Even with data in memory, the first iteration is 7-8 times more expensive than the subsequent ones; when the data are read from GPFS the difference is at two orders of magnitude. We conducted another experiment, where we replaced the local temporary storage for shuffle partitions with GPFS, for the sort-by-key case with 64 nodes. The results showed negligible differences. We also experimented with deploying HDFS and storing temporary data there rather than on local disks directly; such a setting incurred a small overhead of a few seconds. More systematic evaluation of storage alternatives is left for future work.

	10M1000d		100M100d	
partitions	sort	hash	sort	hash
512	1	0.936	0.825	0.81
1024	1.129	1.326	0.88	0.887
2048	1.673	1.829	1.059	1.036

Table III: Impact of the shuffling implementation mechanism (normalized)

E. Additional Spark configurations

Spark contains more than one hundred configuration parameters, and understanding in depth the role and impact of each parameter on performance is an open issue. In this final part of experiments, we aim to review the parameters that have been shown to affect the performance according to the Spark's website or other publications such as [7].

Shuffle manager: Since the version 1.2.0 of Spark, shuffling has ceased to rely on hashing and employs a sort-based approach. In general, sort is expected to perform significantly better for large numbers of partitions, e.g., 15K, relaxing the memory contention for the outgoing buffers. In k-means, we experimented with fewer number of (larger) partitions. In Table III, we show the normalized results for 2 of the datasets, 10M1000d, which suffers from increased data shuffling, and 100M100d. In the former case, there can be performance benefits at the order of 10%, whereas the differences in the latter case are negligible. In that experiment, we employed 32 nodes (64 8-core workers).

Compression: Compression is another important parameter, coming with inherent trade-offs. It may relax memory limitations at the expense of extra computational cost. By default, compression is enabled during shuffling and broadcasting, but is disabled for RDDs. However, to achieve the 23 minutes record for sorting 100TB mentioned earlier, compression has been totally disabled. We further investigate this hint. Regarding k-means, for the 10M1000d dataset, we observed speed-ups of more than 13% (for 32 nodes running 64 8-core workers). When we fully de-activated compression, the memory speed-ups for the 1000M100d were less than 5%. The memory speedups for sort-by-key ranged from approx. 6% (when sorting 1B records with 64 machines) to negligible ones (e.g., when sorting 2B records with 128 machines).

Finally, we have observed that in cases of memory limitation, for example when running k-means on 100GB using 8 nodes, apart from compressions, two additional configurations are beneficial. First, to increase the locality wait threshold; this setting is also used in the 100TB sorting case by Databricks. Second, since k-means does not have any special memory requirements, we can allocate a larger proportion of memory to hold RDDs.

VII. CONCLUSIONS

The research work presented in this paper explores the feasibility and efficiency of deploying Apache Spark over a real-world, petascale, HPC setup, such as MareNostrum. To this end, we have designed and developed a framework, called Spark4MN, to automate the usage of Spark over an IBM LSF-based environment. We have also explored two key algorithms, k-means and sorting, and we showed how we can achieve scalability through proper configuration. Below, we summarize the most important lessons, verified also by additional experiments (not presented here due to space constraints):

First, it is feasible not only to deploy a data-centric paradigm on a compute-centric infrastructure, but also, to achieve high performance in data-intensive applications. Nevertheless, employing a lot of machines and cores does not guarantee high-performance, unless followed by judicious configuration. Second, in our case, k-means, for which data shuffling is not the dominant cost, performed significantly better when we allocated only a single parallel task on each core. For sort-by-key, which is shuffling-intensive, allocating 2 tasks per core, exhibited the highest performance. For even more shuffling-intensive scenarios, the amount of tasks per core needs to be further increased. This correlation between the significance of shuffling in an application and the number of tasks per core applies to a different infrastructure that we tested, too. Third, the size of a spark worker/executor matters, and we observed that having larger workers in terms of the number of cores is beneficial for both types of applications. More specifically, the recommended configuration for Spark4MN has been 8-core workers. For an additional infrastructure tested, this parameter also played a key role and the optimal setting was also application independent (but largely different from the one for Marenostrum). Fourth, GPFS overheads are significant and may outweigh benefits stemming from optimized parallelism configuration; optimized configuration of secondary storage remains an important open issue.

REFERENCES

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” ser. HotCloud’10.

[2] T. White, *Hadoop: The Definitive Guide*, 1st ed. O’Reilly Media, Inc., 2009.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *NSDI*, 2012, pp. 15–28.

[4] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, “Shark: Sql and rich analytics at scale,” in *SIGMOD*, 2013, pp. 13–24.

[5] S. Michael, A. Thota, and R. Henschel, “Hpchadoop: A framework to run hadoop on cray x-series supercomputers,” in *Cray USer Group (CUG)*, 2014.

[6] N. J. Gunther, P. Puglia, and K. Tomasette, “Hadoop superlinear scalability,” *Commun. ACM*, vol. 58, no. 4, pp. 46–55, 2015.

[7] Y. Wang, R. Goldstone, W. Yu, and T. Wang, “Characterization and optimization of memory-resident mapreduce on hpc systems,” in *IPDPS*, 2014, pp. 799 – 808.

[8] S. Jha, J. Qiu, A. Luckow, P. K. Mantha, and G. Fox, “A tale of two data-intensive paradigms: Applications, abstractions, and architectures,” in *IEEE Int. Congress on Big Data*, 2014, pp. 645–652.

[9] A. Marathe, R. Harris, D. K. Lowenthal, B. R. de Supinski, B. Rountree, M. Schulz, and X. Yuan, “A comparative study of high-performance computing on the cloud,” in *HPDC*, 2013, pp. 239–250.

[10] A. Gupta and D. Milojicic, “Evaluation of hpc applications on cloud,” in *Open Cirrus Summit (OCS)*, 2011, pp. 22–26.

[11] H. Herodotou, F. Dong, and S. Babu, “No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics,” in *SOCC*, 2011, pp. 18:1–18:14.

[12] B. Huang, S. Babu, and J. Yang, “Cumulon: optimizing statistical data analysis in the cloud,” in *SIGMOD*, 2013, pp. 1–12.

[13] Y. Kwon, M. Balazinska, B. Howe, and J. A. Rolia, “Skewtune: mitigating skew in mapreduce applications,” in *SIGMOD*, 2012, pp. 25–36.

[14] A. Okcan and M. Riedewald, “Anti-combining for mapreduce,” in *SIGMOD*, 2014, pp. 839–850.

[15] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark, LIGHTNING-FAST DATA ANALYSIS*. O’Reilly Media, 2015.

[16] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, “Scalable k-means++,” *PVLDB*, vol. 5, no. 7, pp. 622–633, 2012.

[17] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Çetintemel, and S. Zdonik, “An architecture for compiling udf-centric workflows,” *PVLDB*, vol. 8, no. 12, pp. 1466–1477, 2015.