Procedia Computer Science

Volume 53, 2015, Pages 121–130

2015 INNS Conference on Big Data

Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf

Jorge L. Reyes-Ortiz[1], Luca Oneto[2], and Davide Anguita[1]

[1] DIBRIS, University of Genoa, Via Opera Pia 13, I-16145, Genoa, Italy
(jorge.reyes.ortiz@smartlab.ws, davide.anguita@unige.it)
[2] DITEN, University of Genoa, Via Opera Pia 11A, I-16145, Genoa, Italy (luca.oneto@unige.it)

Abstract
One of the biggest challenges of the current big data landscape is our inability to process vast amounts of information in a reasonable time. In this work, we explore and compare two distributed computing frameworks implemented on commodity cluster architectures: MPI/OpenMP on Beowulf that is high-performance oriented and exploits multi-machine/multi-core infrastructures, and Apache Spark on Hadoop which targets iterative algorithms through in-memory computing. We use the Google Cloud Platform service to create virtual machine clusters, run the frameworks, and evaluate two supervised machine learning algorithms: KNN and Pegasos SVM. Results obtained from experiments with a particle physics data set show MPI/OpenMP outperforms Spark by more than one order of magnitude in terms of processing speed and provides more consistent performance. However, Spark shows better data management infrastructure and the possibility of dealing with other aspects such as node failure and data replication.

Keywords: Big Data, Supervised Learning, Spark, Hadoop, MPI, OpenMP, Beowulf, Cloud, Parallel Computing

1 Introduction

The information age brings along an explosion of big data from multiple sources in every aspect of our lives: human activity signals from wearable sensors, experiments from particle discovery research and stock market data systems are only a few examples [48]. Recent trends in the area suggest that in the following years the exponential data growth will continue [28], and that there is a strong need to find efficient solutions to deal with aspects such as data storage, real-time processing, information extraction and abstract model generation.

Big data analytics is the area of research focused on collecting, examining and processing large multi-modal and multi-source datasets in order to discover patterns, correlations and extract information from data [48]. This is usually accomplished through the use of supervised and unsupervised machine learning algorithms that learn from the available data. However, these are usually highly computationally expensive, either in the training or prediction phases (e.g. Support Vector Machines (SVM) and K-Nearest Neighbors (KNN) respectively), to the

Selection and peer-review under responsibility of the Scientific Programme Committee of INNS-BigData2015

doi:10.1016/j.procs.2015.07.286

point of becoming intractable using serial algorithm implementations as they are not able to handle current data volumes [43]. Alternatively, parallel approaches have been proposed in order to boost processing speeds [45] but this clearly requires technologies that support distributed computations.

Several technologies are currently available that exploit multiple levels of parallelism (e.g. multi-core, many-core, GPU, cluster, etc) [10, 46, 31]. They trade-off aspects such as performance, cost, failure management, data recovery, maintenance and usability in order to provide solutions adapted to every application. For example, a preference for speed at the expense of a low fault tolerance and vice versa.

Two of the commonly used cluster computing frameworks for big data analytics are: (i) Apache Spark [47], a recently developed platform from UC Berkley that exploits in-memory computation for solving iterative algorithms and can be run in traditional clusters such as Hadoop; and (ii) OpenMP/MPI which efficiently exploits multi-core clusters architectures such as Beowulf, and combines the MPI paradigm with shared memory multiprocessing. The main differences between these two frameworks are fault tolerance support and data replication. Spark deals with them effectively but with a clear impact on speed. Instead OpenMP/MPI provides a solution mostly oriented to high performance computing but susceptible to faults, in particular, if used in commodity hardware. So far, a comparison between these two frameworks has not been investigated.

One of the recent solutions for big data analytics is the use of cloud computing [2] which makes available hundreds or thousands of machines to provide services such as computing and storage. Traditional in-house solutions generally require large investments [19] in hardware and software, and need to be fully exploited in order to be economically sustainable. Instead, cloud platforms, usually hosted by IT companies such as Google, Amazon, and Microsoft, lease services at affordable prices to people and organizations according to their requirements: time, number of machines, machine types, etc. [12].

In this paper we compare two parallelization technologies (Spark on Hadoop and MPI/OpenMP on Beowulf) by running two supervised learning algorithms (KNN and SVM-Pegasos algorithms) for testing system vertical and horizontal scalability. They were deployed in clusters of virtual machines instantiated in the Google Cloud Platform (GCP) [21] using a logic architecture with different configurations: number of virtual machines (N_M), number of cores per machine (N_C) and learning algorithm hyperparameters. The experiments were performed using the Higgs Data Set [42, 5] which contains 11×10^6 samples of simulated signal processes that produce the Higgs bosons.

The paper is organized as follows: in Section 2, we introduce the parallel implementations of the KNN and Pegasos SVM learning algorithms. Then, we describe the frameworks Spark and OpenMP/MPI in Sections 3 and 4 respectively. Moreover, we explain in Section 5 the experiments performed and present their results. Finally, we summarize the obtained results and provide future research directions in Section 6.

2 Big Data Analytics

Let us recall the now-classical supervised learning framework [39] where a set of data $\mathcal{D}_n = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$ is sampled i.i.d. from an unknown distribution μ over $\mathcal{X} \times \mathcal{Y}$ where $\mathcal{X} \in \mathbb{R}^d$ is an input space ($\boldsymbol{x} \in \mathcal{X}$) and $\mathcal{Y} \in \mathbb{R}$ is an output space ($y \in \mathcal{Y}$). In this work, we focus on binary classification problems where $\mathcal{Y} \in \{\pm 1\}$. The purpose of the learning process is to find a function $f : \mathcal{X} \to \mathcal{Y}$ which best approximates μ. There are two main learning methods classes: Lazy Learning (LL) and Eager Leaning (EL) [23, 29]. The former does not create an explicit model f until a test sample $\boldsymbol{x}^c$ is classified and approximates the distribution μ only in the locality of $\boldsymbol{x}^c$. The main disadvantage of LL is that the entire set $\mathcal{D}_n$ needs to be stored in order to perform classification since no abstraction is made before prediction. Nevertheless

LL methods are often exploited because they are easy to implement and mostly parallelizable [3, 7, 20]. EL, instead, learns a compressed model f from $\mathcal{D}_n$ which is a global approximation of μ and do not requires to keep every data sample. This abstraction allows to reduce memory requirements at the expense of an often computationally intensive learning phase [8, 36].

K-Nearest Neighbors (KNN) is one of the most popular LL supervised learning algorithms due to its implementation simplicity and effectiveness [15]. To classify a test sample $\boldsymbol{x}^c$, KNN computes the distances, under some metric, from $\boldsymbol{x}^c$ to each sample $\boldsymbol{x} \in \mathcal{D}_n$ and collects the labels $\boldsymbol{y}^k = \{y_1, \ldots, y_k\}$ of the k nearest samples. The label y^c is found with the mode of $\boldsymbol{y}^k$. Algorithm 1 depicts the pseudocode of KNN classification and highlights the parallelizable parts (data reading and distances computation) and bottlenecks that require serialization (finding the k nearest neighbors and mode calculation). Note that KNN has two hyperparameters: distance metric and k [40]. The Euclidean Distance is usually the preferred metric, at least if d is not too large [34, 40]. Instead, k must be carefully tuned since it deeply influences the algorithm's generalization ability [9], therefore, different values of k are tested over a validation set in order find the one that provides the best classification performance [4].

Regarding EL, several algorithms have been proposed (e.g. Neural Networks, Random Forrest, Support Vector Machines (SVM)) [18]. However, many of them are not easily parallelizable on large scale since they require considarable data exchange between nodes [1]. Iterative learning algorithms such as SVM and Logistic Regression are among the preferred choices in this context, since they offer the advantages of EL with a reasonably parallelizable learning process [46]. The SVM algorithm [39] finds a function in the form $f(\boldsymbol{x}) = \text{sign}(\boldsymbol{w}^T\boldsymbol{x})$ with $\boldsymbol{w} \in \mathbb{R}^d$ that minimizes the following objective function:

$$\boldsymbol{w}: \quad \arg\min_{\boldsymbol{w}} \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{n}\sum_{i=1}^{n} \max[0, 1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i] \tag{1}$$

where $\|\boldsymbol{w}\|^2$ is the regularizer and $\ell(\boldsymbol{w}^T\boldsymbol{x}_i, y_i) = \max[0, 1 - y_i\boldsymbol{w}^T\boldsymbol{x}_i]$ is the hinge loss function. Eq. (1) is a Convex Problem (CP) in which λ balances the trade-off between accuracy and complexity of the solution. λ is for SVM what k is for KNN, and it must be tuned in order to improve the generalization ability of $f(\boldsymbol{x})$. In order to solve the CP, only a few parallel approaches are available [37], including the deterministic sub-gradient descent learning method called Pegasos [35]. In this paper, we adopt the Pegasos SVM [39] approach and present it in Algorithm (2). Each algorithm iteration needs data from its previous one, thus a serial execution is required. However, internal operations such as the estimation of $\mathcal{I}$ and $\boldsymbol{w}$ are easily and partially parallelizable respectively. In this algorithm we only report the learning phase since the classification phase is straightforward and computationally negligible in contrast to training. Note that Pegasos has an additional hyperparameter: the number of iterations Twhich can act as another regularizer. In fact, the optimization process can be stopped before the minimum of the CP is reached (for a fixed value of λ). This concept is called early stopping and has been previously investigated [13].

Algorithm 1: KNN algorithm.

Input: $\mathcal{D}_n$, k and $\{\boldsymbol{x}_1^c, \ldots, \boldsymbol{x}_{n_c}^c\}$
Output: $\{y_1^c, \ldots, y_{n_c}^c\}$

Read $\mathcal{D}_n$; /* Parallelizable */
Read $\{\boldsymbol{x}_1^c, \ldots, \boldsymbol{x}_{n_c}^c\}$; /* Parallelizable */
for $j \leftarrow 1$ **to** n_c **do** /* Parallelizable */
  for $i \leftarrow 1$ **to** n **do** /* Parallelizable */
    $d_i = \text{distance}(\boldsymbol{x}_j^c, \boldsymbol{x}_i)$; /* Parallelizable */
  $\mathcal{I}$ = smallest k elements in $\boldsymbol{d}$; /* Bottleneck */
  $y_i^c = \text{mode}(\{y_i : i \in \mathcal{I}\})$; /* Bottleneck */
return $\{y_1^c, \ldots, y_{n_c}^c\}$;

Algorithm 2: Pegasos SVM.

Input: $\mathcal{D}_n$, λ and number if iterations T
Output: $\boldsymbol{w}$

Read $\mathcal{D}_n$; /* Parallelizable */
$\boldsymbol{w} = 0$;
for $t \leftarrow 1$ **to** T **do** /* Bottleneck */
  $\mathcal{I} = \{i : i \in \{1, \ldots, n\}, y_i\boldsymbol{w}^T\boldsymbol{x}_i < 1\}$; /* Parallelizable */
  $\eta_t = 1/\lambda t$;
  $\boldsymbol{w} = (1 - \eta_t\lambda)\boldsymbol{w}$;
  $\boldsymbol{w}$ += $\eta_t/n \sum_{i\in\mathcal{I}} y_i\boldsymbol{x}_i$; /* Parallelizable/Bottleneck */
return $\boldsymbol{w}$;

3 Spark on Hadoop

Spark is a state-of-the-art framework for high performance parallel computing designed to efficiently deal with iterative computational procedures that recursively perform operations over the same data [47], such as supervised machine learning algorithms. It is based on the concept of maintaining data in memory rather than in disk as it is done by other well-known approaches such as Apache Mahout that require data reloading and incur considerable latencies. Experiments have shown Spark outperforms by up to two orders of magnitude conventional MapReduce jobs in terms of speed [44, 46].

The core data units in Spark are called Resilient Distributed Datasets (RDDs). They are a distributed, immutable and fault-tolerant memory abstraction that collects a set of elements in which a set of operations can be applied to either produce other RDDs (transformations) or return values (actions). RDDs can reside in memory, disk or in combination. However, they are only computed on actions following a Lazy Evaluation (LE) strategy, in order to perform minimal computation and prevent unnecessary memory usage. RDDs are not cached in memory by default, therefore, when data are reused, a *persist* method is needed in order to avoid recomputation.

Various cluster management options are available for running Spark: they range from the simple Spark's integrated Standalone Scheduler to other widespread cluster managers such as Apache Mesos and Hadoop YARN [25]. In this work, we chose to deploy Spark in an Hadoop cluster. Apache Hadoop is an open-source software platform for distributed big data processing over commodity cluster architectures [41]. It has three main elements: a) a MapReduce programming model that separates data processing in *mapping* for performing data operations locally, *suffling* for data redistribution over the network and *reduction* for data summarization; b) a distributed file system (HDFS) with high-throughput data access; and c) a cluster manager (YARN) in charge of handling the available computing resources and job scheduling.

The selected Hadoop architecture was composed of N_M slave machines and two additional machines that run as masters: one for controlling HDFS (namenode) and another for resource management. The software packages installed on each machine were Hadoop 2.4.1 and Spark 1.1.1. In order to tune parallelism and exploit all the machines and cores simultaneously, we set the number of Spark data partitions to be $N_M N_C$ instead of using default values. Moreover, in order to avoid bottlenecks, we verified machine memory requirements to guarantee all the train data was kept in memory and no spill to disk or recalculations occurred. We also used data serialization (Kryo) for faster network performance. Spark provides a machine learning library (MLlib) with a set of standard algorithms such as KMeans, decision tree, logistic regression and some SVM variants, but it does not currently support KNN or Pegasos SVM.

Algorithms 3 and 4 show the pseudocode used for KNN and Pegasos SVM. Both algorithms were written in Spark using Java. They read data from files on HDFS and use standard supported RDD transformations and actions. Notice that Spark operations only need function objects to be passed in order to perform distributed computations over the data. This explains, for example, why there is no loop over the train data map/reduce operations as opposed to Algorithms 1 and 2. The KNN prediction algorithm maps the Euclidean distance from each train sample to a test sample $\boldsymbol{x}_j$ and then returns the k nearest neighbors labels. The predicted class is obtained by computing the mode over these labels. In contrast, Pegasos iterative learning combines a *filter* function that selects the train samples that satisfy $\{i : i \in \{1, \ldots, n\}, y_i \boldsymbol{w}^T \boldsymbol{x}_i < 1\}$ and then the gradient $\boldsymbol{g}$ is obtained by summing individual gradient estimations from the selected samples. Finally the weights are updated with $\boldsymbol{g}$. In both algorithms the training set $\mathcal{D}_n$ is cached in memory as it is recursively used in every iteration.

Algorithm 3: Spark KNN

Input: $\mathcal{D}_n$, k and $\{\boldsymbol{x}_1^c, \ldots, \boldsymbol{x}_{n_c}^c\}$
Output: $\{y_1^c, \ldots, y_{n_c}^c\}$
1 Read $\mathcal{D}_n$
2 Read $\{\boldsymbol{x}_1^c, \ldots, \boldsymbol{x}_{n_c}^c\}$
3 **for** $j \leftarrow 1$ **to** n_c **do**
4 $\quad \{(y_1, d_1), \ldots, (y_k, d_k)\} = D_n$.map(EuclideanDistance($\boldsymbol{x_j}$)). takeordered($k$, DistanceComparator())
5 $\quad y_j^c = \text{mode}(\{y_i\})$
6 **return** $\{y_1^c, \ldots, y_{n_c}^c\}$;

Algorithm 4: Spark Pegasos SVM.

Input: $\mathcal{D}_n$, λ and number if iterations T
Output: $\boldsymbol{w}$
1 Read $\mathcal{D}_n$
2 Set $\boldsymbol{w} = 0$;
3 **for** $t \leftarrow 1$ **to** T **do**
4 $\quad \boldsymbol{g} = \mathcal{D}_n$.filter(GradientCondition($\boldsymbol{w}$)).map(Gradient()). reduce(Sum()) ;
5 $\quad \eta_t = 1/\lambda t$;
6 $\quad \boldsymbol{w} = (1 - \eta_t \lambda)\boldsymbol{w} + \boldsymbol{g}$
7 **return** $\boldsymbol{w}$;

4 MPI/OpenMP on Beowulf

Message Passing Interface (MPI) is a language-independent communications protocol for parallel computing where point-to-point and collective communication are supported [22]. MPI goals are high performance, scalability, and portability. MPI is currently the dominant model used in high-performance computing [38] and is a *de facto* communication standard that provides portability among parallel programs running on distributed memory systems. However, the standard does not currently support fault tolerance [33] since it mainly addresses High-Performance Computing (HPC) problems. Another MPI drawback is that it is not suitable for small grain level of parallelism, for example, to exploit the parallelism of multi-core platforms for shared memory multiprocessing. OpenMP, on the other hand, is an Application Programming Interface (API) that supports multi-platform shared memory multiprocessing programming [16, 14] on most processor architectures and operating systems. OpenMP is becoming the standard for shared memory parallel programming for its high performance, but unfortunately, it is not suitable for distributed memory systems. The idea of extending this API to cope with this issue is now a growing field of research [6]. OpenMP's user-friendly interface allows to easily parallelize complex algorithms. The same thing cannot be said about MPI since the code must be heavily re-engineered in order to obtain relevant performance improvements. However, in 2008 it was proposed a Fortran extension for parallel processing on distributed memory systems called Coarray Fortran (CAF) [30]. Most of the available CAF implementations rely on the MPI standard [24, 17]. Consequently, the combination of these tools, in the Fortran language, allow two levels of granularity: small grain parallelism with OpenMP and large grain parallelism with MPI-based CAF [24, 17]. In particular, we make use of the CAF implementation Intel Parallel Studio XE 2015 together with OpenMP [24]. With this configuration, usually called hybrid OpenMP/MPI HPC on a Beowulf cluster [11, 32, 27], we can fully exploit the multi-core architecture through OpenMP and cluster architecture through CAF. The novelty of our approach is that we can develop and deploy algorithms which take full advantage of tens or thousands of machines with tens of cores. The cluster architecture is quite simple: all the machines are connected through passwordless ssh and one of them has installed Intel Parallel Studio XE 2015 which is accessible from every node. Machines store in equal parts the entire dataset $\mathcal{D}_n$, this is n/N_M data samples per machine (D^i_{n/N_M}, $i \in \{1, \ldots, N_M\}$). N_M MPI processes run in parallel, one for each machine, and every process launches N_C OpenMP threads per machine for maximum architecture usage. Based on the previous considerations we report in Algorithms 5 and 6 the MPI/OpenMP parallel implementation of KNN and Pegasos. The main idea for MPI is to perform all the possible independent parts in parallel without any communication between the processes. Then, machines are synchronized in $O(log(N_M))$ time by exploiting all the available bandwidth with a fast three reduction process. A similar procedure is done with the OpenMP threads. This reduction phase applies for both supervised algorithms at different stages: accumulation of the gradient in Pegasos and search of the k nearest neighbors in KNN. Note that reading $\mathcal{D}^M_{n/N_M}$ from disk cannot take advantages of the multi-core architecture since

there is a bottleneck when reading the disk that may depend on the physical implementation of the computing center.

Algorithm 5: MPI/OpenMP KNN.

Input: N_M, $\mathcal{D}^1_{n/N_M}, \ldots, \mathcal{D}^{N_M}_{n/N_M}$, k and $\{\boldsymbol{x}^c_1, \ldots, \boldsymbol{x}^c_{n_c}\}$
Output: $\{y^c_1, \ldots, y^c_{n_c}\}$

launch N_M MPI processes, one for each machine;
each machine $M \in \{1, \ldots, N_M\}$ **begin**
  Read $\mathcal{D}^M_{n/N_M}$;
  Read $\{\boldsymbol{x}^c_1, \ldots, \boldsymbol{x}^c_{n_c}\}$;
  for $j \leftarrow 1$ **to** n_c **do**
    launch N_C OpenMP threads;
    Each Core $C \in \{1, \ldots, N_C\}$ **begin**
      for $i = C : C : n/N_M$ **do**
        $d_i = \text{distance}(\boldsymbol{x}^c_j, \boldsymbol{x}_i)$;
    $\mathcal{I}^M$ = **tree reduction process** {**smallest k elements**} $\in \boldsymbol{d}$;
    close all the N_C OpenMP threads;
    $\mathcal{I}$ = **tree reduction process** {**smallest k elements**} $\in \{\mathcal{I}^1, \ldots, \mathcal{I}^M\}$;
    wait all the N_M MPI processes;
    if $M == 1$ **then**
      $y^c_i = \text{mode}(\{y_i : i \in \mathcal{I}\})$;
  if $M == 1$ **then**
    return $\{y^c_1, \ldots, y^c_{n_c}\}$;
close all the N_M MPI processes;

Algorithm 6: MPI/OpenMP Pegasos SVM.

Input: $\mathcal{D}_n$, λ and number if iterations T
Output: $\boldsymbol{w}$

launch N_M MPI processes, one for each machine;
each machine $M \in \{1, \ldots, N_M\}$ **begin**
  Read $\mathcal{D}^M_{n/N_M}$;
  $\boldsymbol{w} = 0$;
  for $t \leftarrow 1$ **to** T **do**
    launch N_C OpenMP threads;
    Each Core $C \in \{1, \ldots, N_C\}$ **begin**
      $\boldsymbol{a}^C = 0$;
      $\mathcal{I} = \{i : i \in \{C : C : n\}, y_i \boldsymbol{w}^T \boldsymbol{x}_i < 1\}$;
      $\eta_t = 1/\lambda t$;
      $\boldsymbol{a}^C = \eta_t / n \sum_{i \in \mathcal{I}} y_i \boldsymbol{x}_i$;
    $\boldsymbol{b}^M$ = **tree reduction process** {+} **over** $\{\boldsymbol{a}^1, \ldots, \boldsymbol{a}^C\}$;
    close all the N_C OpenMP threads;
    $\boldsymbol{b}$ = **tree reduction process** {+} **over** $\{\boldsymbol{b}^1, \ldots, \boldsymbol{b}^M\}$;
    wait all the N_M MPI process;
    $\boldsymbol{w} = (1 - \eta_t \lambda)\boldsymbol{w} + \boldsymbol{b}$;
  return $\boldsymbol{w}$;
close all the N_M MPI processes;

5 Evaluation

In this section we show the results of the Spark on Hadoop and MPI/OpenMP on Beowulf implementations of KNN and Pegasus SVM algorithms with different cluster configurations. We run on GCP Linux shell script routines for automatically deploying the clusters of virtual machines from the ground up with the selected configurations, including operating system (latest version of CentOS 6) and software installation. We tested KNN Algorithm 3 on Spark and Algorithm 5 on MPI/OpenMP for different values of $k \in \{1, 10, 100\}$, with different number of machines $N_M \in \{5, 10, 20\}$ and number of cores per machine $N_C \in \{4, 8, 16\}$. To this end, we used the *n1-standard-4*, *n1-standard-8* and *n1-standard-16* machine types from GCP with 15, 30 and 60 GB of ram and 4, 8 and 16 cores respectively. Regarding non-volatile memory, each machine was equipped with a 500 GB SSD disk. Every combination of parameters was run three times. Similarly, we tested Pegasos Algorithms 4 and 6, only differing in the hyperparameters with $\lambda \in \{0.01, 1, 100\}$ and $T = 100$.

We exploited the HIGGS Data Set [42], available from the UCI Machine Learning Repository [26], that aims to discriminate between signal and background processes that produce Higgs bosons. Data consist of 11000000 samples in 28 dimensions which were produced using Monte Carlo simulations [5]. The first 21 features are kinematic properties measured by the particle detectors in the accelerator and the last 7 are high-level features derived by physicists from the first group to help in the classification. The last 500000 examples were used as a test set. Test and train data were stored in separated text files for the experiment and the amount of disk space needed was 7GB. The dataset size allowed seeing the scalability break point of one the technologies (Spark) while the other continued to scale. Moreover, the dataset size was selected to fit entirely in the memory of the smallest cluster configuration according to the virtual machines specifications. We avoided data reloading and spilling to disk as these conditions would have not produced a fully in-memory application and yielded to more time-consuming results. We also took into account that Spark allocates memory on each executor for data storage, data shuffling and memory heap, reducing the amount of memory for keeping the dataset in memory. Moreover, Spark executors memory needed to fit inside Hadoop's containers which are also constrained by other running applications such as the operating system on each virtual machine.

As a result of Spark's LE nature, the time to read the data from disk was measured together with the first action over RDDs. This coincides with the reductions over the train data. For this reason, we measured two different times in our experiments: disk reading together with the first MapReduce operation, and the average time of the remaining iterations. Similar time evaluations have been used in the literature such as in [46] and, for comparison purposes, we also performed the same measurements on MPI/OpenMP. In Table 1 KNN implementation results are presented. They include the following quantities:

- TS: theoretical speedup (for a fixed k with respect to the smallest cluster: $N_M = 5$ and $N_C = 4$).
- $\Delta_t^{1^0}$: time to read the data and classify the first sample in seconds.
- $\Delta_t^{2^0 \cdots}$: the average time to classify the remaining samples in milliseconds.
- $S1$, $S2$: the data reading and classification speedups of each cluster with respect to the smallest cluster.
- $S\Delta_t^{1^0}$ the speedup of MPI/OpenMP against Spark to classify the first sample.
- $S\Delta_t^{2^0 \cdots}$ the average speedup of MPI/OpenMP against Spark to classify the remaining samples.

Conversely, we report in Table 2 the results of Pegasos SVM (Algorithms 4 and 6). In this case $\Delta_t^{1^0}$ is the time to perform the first learning iteration and $\Delta_t^{2^0 \cdots}$ is the average time of the subsequent iterations.

From Tables 1 and 2 it is possible to draw some conclusions about the performance of the approaches over the HIGGS dataset. In particular:

- From a computational point of view, the MPI/OpenMP implementation is much more powerful than the Spark on Hadoop alternative in terms of speed. In this particular application, it can be more than 10 times faster.
- MPI/OpenMP scales better than Spark. This is mainly to its low level programming language and reduced overhead (e.g. no fault handling such as in Spark). The speedup of MPI/OpenMP over Spark is due to the dataset size with respect to the cluster size. With a larger dataset, time differences between the two implementations would be smaller. This is explained due to an increase of the effective computation time (e.g. classification) with respect to the overhead of the technologies.
- For what concerns to data I/O management, MPI/OpenMP and Spark are much closer in terms of disk access time although MPI/OpenMP is still faster. Moreover, Spark on Hadoop is better suited to exploit the GCP infrastructure since it is able to split the data in multiple chunks so files can be read in parallel from many sectors. This is because in GCP, virtual disks can spread over multiple physical disks so to improve the I/O performance. Our MPI/OpenMP implementation does not currently exploit this possibility. Future solutions will address this issue.

6 Conclusions

As a concluding remark, we underline that even if Spark on Hadoop with in-memory data processing reduces the gap between Hadoop MapReduce and HPC for Machine Learning, we are still far from achieving state-of-the-art HPC technologies performance. Nevertheless, Spark on Hadoop may be preferred because it also:

- offers a distributed file system with failure and data replication management.
- allows the addition of new nodes at runtime.
- provides a set of tools for data analysis and management that is easy to use, deploy and maintain.

So far, an integration between Hadoop and MPI/OpenMP has not been proposed. This idea is an interesting subject of research because it could greatly improve speed performance, an

Algorithms				Spark (Alg 3)				OpenMP/MPI (Alg 5)				Comparison	
k	N_C	N_M	TS	$\Delta_t^{1^0}$ (s)	$\Delta_t^{2^{0\cdots}}$ (ms)	S1	S2	$\Delta_t^{1^0}$ (s)	$\Delta_t^{2^{0\cdots}}$ (ms)	S1	S2	$S\Delta_t^{1^0}$	$S\Delta_t^{2^{0\cdots}}$
1	4	5	1	36.07 ±1.54	189.65 ±13.49	1	1	3.2 ±0.05	18.50 ±0.43	1	1	11.28	10.25
1	4	10	2	23.82 ±1.54	124.99 ±14.99	1.51	2.61	1.6 ±0.02	10.01 ±0.21	2	1.4	14.9	12.49
1	4	20	4	13.39 ±2.77	91.66 ±13.11	2.69	3.56	0.8 ±0.01	5.43 ±0.13	3.99	2.58	16.72	16.89
1	8	5	2	22.48 ±1.58	123.86 ±09.66	1.6	2.64	3.2 ±0.05	9.37 ±0.22	1	1.49	7.02	13.22
1	8	10	4	12.43 ±2.33	87.73 ±16.67	2.9	3.72	1.6 ±0.02	5.11 ±0.11	2	2.74	7.76	17.18
1	8	20	8	9.58 ±2.25	76.32 ±19.85	3.76	4.28	0.8 ±0.01	2.90 ±0.06	3.99	4.82	11.95	26.29
1	16	5	4	14.52 ±2.91	92.57 ±14.55	2.48	3.53	3.2 ±0.05	4.91 ±0.10	1	2.85	4.54	18.84
1	16	10	8	9.01 ±0.61	69.44 ±12.79	4	4.7	1.6 ±0.02	2.89 ±0.07	2	4.84	5.64	23.99
1	16	20	16	7.45 ±1.95	76.52 ±16.03	4.84	4.27	0.8 ±0.01	1.56 ±0.04	3.99	8.95	9.29	48.93
10	4	5	1	34.73 ±1.02	209.32 ±17.40	1	1	3.2 ±0.05	31.37 ±0.67	1	1	10.87	6.67
10	4	10	2	24.56 ±1.20	150.34 ±13.15	1.41	2.09	1.6 ±0.03	19.79 ±0.41	2	1.21	15.36	7.6
10	4	20	4	14.63 ±2.44	102.30 ±16.10	2.37	3.08	0.8 ±0.01	9.28 ±0.20	3.99	2.59	18.25	11.02
10	8	5	2	22.12 ±0.32	137.79 ±09.59	1.57	2.29	3.2 ±0.04	15.96 ±0.34	1	1.5	6.91	8.63
10	8	10	4	13.52 ±2.61	99.05 ±14.51	2.57	3.18	1.6 ±0.02	9.42 ±0.22	2	2.55	8.45	10.51
10	8	20	8	9.85 ±2.24	82.26 ±20.41	3.52	3.83	0.8 ±0.01	5.14 ±0.11	4	4.67	12.32	16.01
10	16	5	4	13.25 ±2.11	104.08 ±12.19	2.62	3.03	3.2 ±0.05	8.18 ±0.19	1	2.93	4.15	12.72
10	16	10	8	9.12 ±2.72	74.90 ±11.73	3.81	4.2	1.6 ±0.02	4.24 ±0.10	2	5.66	5.7	17.67
10	16	20	16	7.31 ±1.36	85.15 ±16.79	4.75	3.7	0.8 ±0.01	2.29 ±0.05	3.99	10.47	9.13	37.16
100	4	5	1	35.62 ±0.68	214.27 ±15.12	1	1	3.2 ±0.05	130.23 ±2.81	1	1	11.11	1.65
100	4	10	2	23.64 ±1.21	153.51 ±16.04	1.52	2.28	1.6 ±0.02	66.46 ±1.62	2	1.5	14.78	2.31
100	4	20	4	15.66 ±0.78	113.64 ±14.19	2.29	3.08	0.8 ±0.01	35.25 ±0.77	4.01	2.84	19.6	3.22
100	8	5	2	22.30 ±2.52	186.39 ±96.02	1.61	1.88	3.2 ±0.04	65.27 ±1.47	1	1.53	6.97	2.86
100	8	10	4	10.98 ±0.36	136.11 ±13.11	3.27	2.57	1.6 ±0.02	34.13 ±0.73	2	2.93	6.87	3.99
100	8	20	8	9.18 ±2.38	102.86 ±29.72	3.91	3.4	0.8 ±0.01	18.44 ±0.42	4	5.42	11.48	5.58
100	16	5	4	14.31 ±1.30	144.74 ±83.47	2.51	2.42	3.2 ±0.05	32.72 ±0.79	1	3.06	4.46	4.42
100	16	10	8	10.96 ±3.94	91.55 ±11.73	3.28	3.82	1.6 ±0.02	17.12 ±0.40	2	5.84	6.83	5.35
100	16	20	16	7.32 ±2.18	111.02 ±17.71	4.9	3.15	0.8 ±0.01	8.88 ±0.20	3.99	11.26	9.14	12.5

Table 1: KNN Results

Algorithms				Spark (Alg 4)				OpenMP/MPI (Alg 6)				Comparison	
λ	N_C	N_M	TS	$\Delta_t^{1^0}$ (s)	$\Delta_t^{2^{0\cdots}}$ (ms)	S1	S2	$\Delta_t^{1^0}$ (s)	$\Delta_t^{2^{0\cdots}}$ (ms)	S1	S2	$S\Delta_t^{1^0}$	$S\Delta_t^{2^{0\cdots}}$
0.01	4	5	1	37.58 ±0.53	446.83 ±54.33	1	1	3.2 ±0.04	34.16 ±0.41	1	1	11.74	13.08
0.01	4	10	2	24.91 ±1.12	396.48 ±55.05	1.51	1.49	1.6 ±0.02	18.58 ±0.22	2	1.4	15.56	21.34
0.01	4	20	4	13.30 ±1.92	240.87 ±50.67	2.83	2.46	0.8 ±0.01	9.46 ±0.13	4	2.75	16.62	25.45
0.01	8	5	2	22.34 ±1.23	321.92 ±36.96	1.68	1.84	3.2 ±0.04	17.08 ±0.23	1	1.52	6.98	18.85
0.01	8	10	4	13.62 ±1.87	214.02 ±44.51	2.76	2.76	1.6 ±0.02	9.09 ±0.11	2	2.86	8.51	23.55
0.01	8	20	8	10.42 ±2.28	162.93 ±25.80	3.61	3.63	0.8 ±0.01	4.92 ±0.07	4	5.28	13.02	33.11
0.01	16	5	4	13.79 ±0.29	249.50 ±28.15	2.73	2.37	3.2 ±0.04	8.72 ±0.12	1	2.98	4.31	28.61
0.01	16	10	8	10.72 ±1.18	171.79 ±18.69	3.5	3.44	1.6 ±0.02	4.80 ±0.06	2	5.42	6.7	35.83
0.01	16	20	16	7.74 ±2.54	126.24 ±26.57	4.85	4.69	0.8 ±0.01	2.48 ±0.03	4	10.5	9.67	50.96
1	4	5	1	35.99 ±0.54	631.54 ±73.26	1	1	3.2 ±0.04	34.17 ±0.43	1	1	11.23	18.48
1	4	10	2	24.75 ±1.74	599.03 ±66.56	1.47	1.25	1.6 ±0.02	18.44 ±0.25	2	1.41	15.45	32.49
1	4	20	4	14.65 ±2.41	339.13 ±87.73	2.49	2.2	0.8 ±0.01	9.49 ±0.12	4	2.74	18.29	35.72
1	8	5	2	23.98 ±0.86	471.04 ±36.05	1.52	1.58	3.2 ±0.04	17.07 ±0.24	1	1.52	7.49	27.59
1	8	10	4	12.98 ±2.01	263.94 ±26.33	2.81	2.83	1.6 ±0.02	8.87 ±0.12	2	2.93	8.1	29.76
1	8	20	8	8.04 ±0.77	207.26 ±28.70	4.54	3.6	0.8 ±0.01	5.06 ±0.07	4	5.13	10.04	40.93
1	16	5	4	14.60 ±2.46	335.65 ±28.07	2.5	2.22	3.2 ±0.03	8.79 ±0.11	1	2.96	4.56	38.18
1	16	10	8	9.80 ±3.06	218.49 ±14.99	3.72	3.41	1.6 ±0.02	4.89 ±0.07	2	5.32	6.12	44.72
1	16	20	16	7.67 ±1.11	160.14 ±20.48	4.76	4.66	0.8 ±0.01	2.72 ±0.04	4	9.56	9.58	58.85
100	4	5	1	37.35 ±2.20	740.64 ±64.61	1	1	3.2 ±0.04	33.98 ±0.45	1	1	11.67	21.8
100	4	10	2	24.87 ±2.47	542.16 ±32.44	1.5	1.37	1.6 ±0.02	18.39 ±0.23	2	1.41	15.54	29.47
100	4	20	4	16.33 ±0.67	343.26 ±61.95	2.29	2.16	0.8 ±0.01	9.50 ±0.12	4	2.74	20.39	36.11
100	8	5	2	24.24 ±0.92	469.63 ±34.37	1.54	1.58	3.2 ±0.04	17.16 ±0.23	1	1.52	7.57	27.37
100	8	10	4	13.08 ±1.93	220.90 ±27.98	2.86	3.35	1.6 ±0.02	9.42 ±0.13	2	2.76	8.17	23.46
100	8	20	8	9.53 ±2.14	215.23 ±24.74	3.92	3.44	0.8 ±0.01	4.80 ±0.06	4	5.41	11.91	44.82
100	16	5	4	17.26 ±0.77	425.25 ±17.40	2.16	1.74	3.2 ±0.04	8.61 ±0.11	1	3.02	5.39	49.41
100	16	10	8	9.63 ±1.37	262.48 ±64.61	3.88	2.82	1.6 ±0.02	4.90 ±0.06	2	5.31	6.02	53.59
100	16	20	16	7.72 ±1.65	159.01 ±22.78	4.84	4.66	0.8 ±0.01	2.85 ±0.04	4.01	9.12	9.66	55.77

Table 2: Pegasos Results

aspect many research/industry sectors are really interested in, even at the expense of a weak failure management. Future work will also consider the comparison of the technologies over larger datasets, in particular, when the data cannot be fully kept in memory.

References

[1] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford. A reliable effective terascale linear learning system. *The Journal of Machine Learning Research*, 15(1):1111–1133, 2014.

[2] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing: current state and future opportunities. In *International Conference on Extending Database Technology*, pages 530–533, 2011.

[3] D. W. Aha. *Lazy learning*. Kluwer academic publishers, 1997.

[4] D. Anguita, A. Ghio, L. Oneto, and S. Ridella. In-sample and out-of-sample model selection and error estimation for support vector machines. *IEEE Transactions on Neural Networks and Learning Systems*, 23(9):1390–1406, 2012.

[5] P Baldi, P Sadowski, and D Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5, 2014.

[6] A. Basumallik, S. J. Min, and R. Eigenmann. Programming distributed memory sytems using openmp. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, 2007.

[7] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *ACM International conference on Machine learning*, pages 97–104, 2006.

[8] C. M. Bishop. *Neural networks for pattern recognition*. Clarendon press Oxford, 1995.

[9] O. Bousquet and A. Elisseeff. Stability and generalization. *The Journal of Machine Learning Research*, 2:499–526, 2002.

[10] L. J. Cao, S. S. Keerthi, C. J. Ong, J. Q. Zhang, U. Periyathamby, X. J. Fu, and H. P. Lee. Parallel sequential minimal optimization for the training of support vector machines. *IEEE Transactions on Neural Networks*, 17(4):1039–1049, 2006.

[11] F. Cappello and D. Etiemble. Mpi versus mpi+ openmp on the ibm sp for the nas benchmarks. In *ACM/IEEE Conference on Supercomputing*, pages 12–12, 2000.

[12] A. G. Carlyle, S. L. Harrell, and P. M. Smith. Cost-effective hpc: The community or the cloud? In *IEEE International Conference on Cloud Computing Technology and Science*, pages 169–176, 2010.

[13] R. Caruana, S. Lawrence, and G. Lee. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in Neural Information Processing Systems*, pages 402–410, 2001.

[14] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*. MIT press, 2008.

[15] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[16] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[17] A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle, and D. Rouson. Opencoarrays: open-source transport layers supporting coarray fortran compilers. In *International Conference on Partitioned Global Address Space Programming Models*, pages 4–14, 2014.

[18] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim. Do we need hundreds of classifiers to solve real world classification problems? *The Journal of Machine Learning Research*, 15(1):3133–3181, 2014.

[19] H. Furuta, T. Kameda, Y. Fukuda, and D. M. Frangopol. Life-cycle cost analysis for infrastructure systems: Life cycle cost vs. safety level vs. service life. *Life-cycle performance of deteriorating structures: Assessment, design and management*, pages 19–25, 2004.

[20] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using gpu. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6, 2008.

[21] Google. Google cloud platform - google compute engine. https://cloud.google.com, 2015.

[22] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[23] I. Hendrickx and A. Van Den Bosch. Hybrid algorithms with instance-based classification. In *Machine Learning: European Conference on Machine Learning*, pages 158–169, 2005.

[24] Intel. Intel parallel studio xe 2015 sp2. https://software.intel.com/en-us/intel-parallel-studio-xe, 2015.

[25] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark*. O'Reilly Media, 2015.

[26] M. Lichman. UCI machine learning repository, 2013.

[27] E. Lusk and A. Chan. Early experiments with the openmp/mpi hybrid programming model. In *OpenMP in a New Era of Parallelism*, pages 36–47, 2008.

[28] S. Mills, S. Lucas, L. Irakliotis, M. Rappa, T. Carlson, and B. Perlowitz. Demystifying big data: a practical guide to transforming the business of government. In *Technical report. http://www. ibm. com/software/data/demystifying-big-data*, pages 1–100, 2012.

[29] T. M. Mitchell. *Machine learning. WCB.* McGraw-Hill Boston, 1997.

[30] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, pages 1–31, 1998.

[31] K. Olukotun. Beyond parallel programming with domain specific languages. In *Symposium on Principles and practice of parallel programming*, pages 179–180, 2014.

[32] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 427–436, 2009.

[33] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. R. de Supinski, N. Maruyama, and S. Matsuoka. Fmi: Fault tolerant messaging interface for fast and transparent recovery. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1225–1234, 2014.

[34] D. Schnitzer and A. Flexer. Choosing the metric in high–dimensional spaces based on hub analysis. In *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, pages 1–6, 2014.

[35] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter. Pegasos: Primal estimated sub-gradient solver for svm. *Mathematical programming*, 127(1):3–30, 2011.

[36] J. Shawe-Taylor and N. Cristianini. *Kernel methods for pattern analysis.* Cambridge university press, 2004.

[37] J. Shawe-Taylor and S. Sun. A review of optimization methodologies in support vector machines. *Neurocomputing*, 74(17):3609–3618, 2011.

[38] S. Sur, M. J. Koop, and D. K. Panda. High-performance and scalable mpi over infiniband with reduced memory usage: an in-depth performance analysis. In *ACM/IEEE conference on Supercomputing*, page 105, 2006.

[39] V. N. Vapnik. *Statistical learning theory.* Wiley-Interscience, 1998.

[40] K. Q. Weinberger and L. K. Saul. Distance metric learning for large margin nearest neighbor classification. *The Journal of Machine Learning Research*, 10:207–244, 2009.

[41] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[42] Daniel Whiteson. Higgs data set. `https://archive.ics.uci.edu/ml/datasets/HIGGS`, 2014.

[43] X. Wu, X. Zhu, G. Q. Wu, and W. Ding. Data mining with big data. *IEEE Transactions on Knowledge and Data Engineering*, 26(1):97–107, 2014.

[44] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *ACM SIGMOD International Conference on Management of data*, pages 13–24, 2013.

[45] Y. You, S. L. Song, H. Fu, A. Marquez, M. M. Dehnavi, K. Barker, K. W. Cameron, A. P. Randles, and G. Yang. Mic-svm: Designing a highly efficient support vector machine for advanced modern multi-core and many-core architectures. In *IEEE International Parallel and Distributed Processing Symposium*, pages 809–818, 2014.

[46] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX Conference on Networked Systems Design and Implementation*, pages 2–2, 2012.

[47] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[48] Yiteng Zhai, Y Ong, and I Tsang. The emerging big dimensionality. *IEEE Computational Intelligence Magazine*, 9(3):14–26, 2014.