# Ignis: an efficient and scalable multi-language Big Data framework*

César Piñeiro[a], Rodrigo Martínez-Castaño[a] and Juan C. Pichel[a,*]

[a]CiTIUS, Universidade de Santiago de Compostela, 15782 Santiago de Compostela, Spain

ARTICLE INFO

*Keywords*:
Big Data; Multi-language; Performance; Scalability; Container

Abstract

Most of the relevant Big Data processing frameworks (e.g., Apache Hadoop, Apache Spark) only support JVM (Java Virtual Machine) languages by default. In order to support non-JVM languages, subprocesses are created and connected to the framework using system pipes. With this technique, the impossibility of managing the data at thread level arises together with an important loss in the performance. To address this problem we introduce Ignis, a new Big Data framework that benefits from an elegant way to create multi-language executors managed through an RPC system. As a consequence, the new system is able to execute natively applications implemented using non-JVM languages. In addition, Ignis allows users to combine in the same application the benefits of implementing each computational task in the best suited programming language without additional overhead. The system runs completely inside Docker containers, isolating the execution environment from the physical machine. A comparison with Apache Spark shows the advantages of our proposal in terms of performance and scalability.

## 1. Introduction

Nowadays we are living in the Big Data era, which demands processing in a efficient way huge amounts of data coming from many different sources. In the past, clusters were reserved for HPC (High-Performance Computing) tasks, but with the arrival of Big Data it became necessary to design new frameworks and technologies for the execution on these systems. The *de facto* standards for parallel processing of Big Data are Apache Hadoop [31] and Apache Spark [35] engines. To fully exploit the capabilities of these frameworks programmers should implement their applications in languages based on the Java Virtual Machine (JVM) (fundamentally, Java and Scala) or other high-level languages such as Python.

However, developers build applications in the programming languages that best suit their needs and, many times, those languages are not natively supported by the corresponding Big Data framework. For instance, most of the existent scientific applications are developed in languages like C, C++ or Fortran. In that case, it is necessary to port the source codes, which requires a huge effort, use source-to-source compilers [27] or take advantage of the mechanisms provided by the frameworks to call external processes based on system pipes with the corresponding degradation in the performance [9]. Note that in the latter case, the main code that performs the calls should be anyway implemented in Python, Java or Scala.

To overcome the above limitations of the Big Data processing engines we introduce Ignis, a new framework that allows users to execute their applications written in multiple programming languages without additional overhead. Our framework uses a multi-language RPC (Remote Procedure Call) approach to create a native executor for each language in order to avoid data transfers. In this way, data is handled by the executor in the most efficient way for each programming language. Currently Ignis supports Python, Java, C and C++, but thanks to its modular architecture, adding support for new languages is a straightforward process.

Several efforts have been done to bridge the gap between HPC and Big Data technologies [11, 12, 19, 20, 24, 33]. However, Ignis follows a different path in order to reach this convergence since none of the previous works have as their final goal searching for a unique processing engine for Big Data and HPC applications.

We can summarize the key contributions of this paper as follows:

- To the best of our knowledge, Ignis is the first Big Data framework with native multi-language support including both JVM and non-JVM-based languages. In this way, users might combine in the same application the benefits of implementing each computational task in the best suited programming language.
- Ignis outperforms the state-of-the-art framework Spark in terms of performance and scalability running applications that represent the most typical algorithmic patterns in Big Data and scientific computing. As a consequence, Ignis facilitates the convergence of HPC and Big Data since data and compute-intensive tasks can be executed efficiently in the same framework.
- Contrary to the developers community belief, we show that Python is not natively supported by Spark since data transfers between the JVM and external processes degrade noticeably the overall performance.
- Ignis provides a simple and powerful user API to develop applications. To facilitate the adoption from the Big Data community, the Ignis API was inspired by

*This work has been supported by MICINN (RTI2018-093336-B-C21), Xunta de Galicia (ED431G/08 and ED431C-2018/19) and European Regional Development Fund (ERDF).

*Corresponding author
cesaralfredo.pineiro@usc.es (C. Piñeiro);
rodrigo.martinezl@usc.es (R. Martínez-Castaño);
juancarlos.pichel@usc.es (J.C. Pichel)
ORCID(s): 0000-0001-9505-6493 (J.C. Pichel)

the Spark API in such a way that Ignis codes are easily understandable by users who are familiar with Spark.

– Finally, Ignis is fully developed inside Docker [10] containers, which isolates the execution environment from the physical system and avoids dependency problems. It includes a custom resource manager to assign hardware resources and launch containers.

The remainder of this paper is organized as follows. Section 2 gives the background and discusses some related work. Section 3 describes the architecture and the modules of Ignis. The different ways to storage data in Ignis are explained in Section 4. Section 5 details the Ignis API. Section 6 shows the experimental results. Finally, the main conclusions derived from this work are explained.

## 2. Background & Related Work

### 2.1. Big Data Frameworks

MapReduce [8] is a programming model introduced by Google for processing and generating large data sets on a huge number of computing nodes. A MapReduce program execution is divided into two main phases: *map* and *reduce*. The input and output of a MapReduce computation is a list of key-value pairs. Users only need to focus on implementing map and reduce functions. In the map phase, map workers take as input a list of key-value pairs and generate a set of intermediate output key-value pairs, which are stored in the intermediate storage (i.e., files or in-memory buffers). The reduce function processes each intermediate key and its associated list of values to produce a final dataset of key-value pairs. In this way, map tasks achieve data parallelism, while reduce tasks perform parallel reduction. Currently, several processing frameworks support this programming model such as Hadoop [31], Spark [35], Flink [5] and Tez [28].

In particular, Apache Hadoop is the most successful open-source implementation of the MapReduce programming model. Hadoop consists of three main layers: a data storage layer (HDFS), a resource manager layer (YARN), and a data processing layer (Hadoop MapReduce Framework). HDFS is a block-oriented distributed file system based on the idea that the most efficient data processing pattern is a write-once, read-many-times pattern.

Apache Spark was designed to overcome some of the Hadoop limitations, especially when considering iterative jobs. It supports both in-memory and on-disk computations in a fault tolerant manner by introducing the idea of Resilient Distributed Datasets (RDDs). An RDD represents a read-only collection of objects partitioned across the cluster nodes that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. By using RDDs, programmers can perform iterative operations on their data without writing intermediary results to disk. Apart from running interactively using Python, Scala and R, Spark can also be linked into applications in either Java, Python, Scala or R.

Hadoop and Spark are the *de facto* standards for Big Data processing. Jobs on both frameworks are composed by a driver and a number of executors. The driver is a high-level process that controls the workflow, and executors are a set of independent processes distributed on a cluster that run the work in parallel. The vast majority of the applications developed for these frameworks are written in Java, Scala or Python. However, there are some situations where it is not reasonable to port an application to one of the previous languages (e.g., performance issues). Note that Python is a non-JVM language, so it is not natively supported by Spark. However, Spark takes advantage of Jython [15], which allows a driver implemented in Python to be executed within the JVM. This causes a misunderstanding in the Spark users community. By contrast, executors are directly executed with the available Python interpreter. Hadoop and Spark use system pipes to share data outside of the Java Virtual Machine in order to run non-JVM codes, which introduces an additional overhead that negatively affects performance [9]. Both Hadoop and Spark deal with non-JVM codes in a similar way. Next the process followed by Spark is explained:

– The user application (non-JVM code) must read data from the standard input and write results to the standard output. Therefore, input and output must be represented in a string format.

– An executor containing the input data is created inside a JVM.

– Each executor launches a subprocess with the user application, connecting the JVM to the subprocess using pipes.

– The executor converts each object stored inside an RDD to its string representation, writing the result to the standard input of the subprocess. At the same time, another thread is reading from the standard output of the subprocess to generate the resulting RDD with the output data.

As we mentioned, the above process requires that input and output data must be represented as a string. As a consequence, the user application is responsible to parse this string data to a native format supported by the considered non-JVM language. Note that this process becomes complicated when more complex data structures such as *trees* or *maps* are considered. Another important drawback is that non-JVM processes are not allowed to access Spark and Hadoop functions such as the context.

### 2.2. Bridging the Gap between HPC and Big Data

There is unanimity in the research community about the importance of approaching HPC and Big Data worlds. We can find in the literature several works dealing with this challenge from different points of view. We can categorized those works depending on if the path of convergence goes from Big Data to HPC or in the opposite direction. In this way, the first category includes solutions that try to boost

the performance of well-established Big Data technologies when running on HPC systems. For example, taking advantage of the fast interconnection networks such as Infiniband [19, 21], introducing new solutions for data analytics based on classical HPC programming languages [24], or implementing highly optimized libraries to run on HPC systems but using JVM-based languages [11]. In the second category we find works that extend HPC technologies in order to support Big Data tasks. For instance, extending MPI to improve the processing and communication of large numbers of key-value pairs [20]. Note that, unlike previous works, our framework covers both categories since its final goal is that applications belonging to HPC and Big Data worlds can be executed efficiently in a unique framework.

Finally, there is an interesting paper that do not fit in the previous categories that analyzes and compares in depth the Big Data and HPC software stacks [12]. It identifies several architecture layers where there is synergy, which can facilitate the integration of both stacks.

### 2.3. Apache Thrift

Apache Thrift [1] is an RPC (Remote Procedure Call) system whose main functionality is the invocation of remote methods between different programming languages. Apache Thrift has its own IDL (Interface Definition Language) to define multiple services. In the first place, each service exports series of functions with parameters, returns and exceptions. Second, Thrift generates the corresponding *skeleton* interface and *stub* class for the user selected language. A client uses the *stab* to make remote calls and the server defines the methods implementing the *skeleton*.

Thrift is composed of a set of protocols and transports. Protocols define how data types are serialized, while transports indicate the medium through which the data is sent. There is also the possibility of use intermediate transports such as Zlib [16] to apply compression in streaming when data is sent.

Spark and Ignis use Thrift for the communication between modules, but Ignis uses a modified version to add data transfer without defining an IDL.

### 2.4. Docker

Docker [10] containers provide the benefits of virtualization (isolation, flexibility, portability, agility, etc.) without penalizing the I/O performance considerably. Docker makes use of resource isolation characteristics of the Linux kernel, so independent containers can be executed on the same host using different assigned resources without interfering among them. Containers supply a virtual environment with their own space of processes and networks. The containers are built with stacked layers. When a container is in execution, a new writable layer is created over a set of read-only layers which define a Docker image. The Docker images are always built from a base image, usually a root filesystem coming from a GNU/Linux distribution. These images can be easily distributed via the official registry, with our own registry or with *tarballs*. Images can be built with a custom scripting language (*dockerfiles*) or by saving the state of a running container.

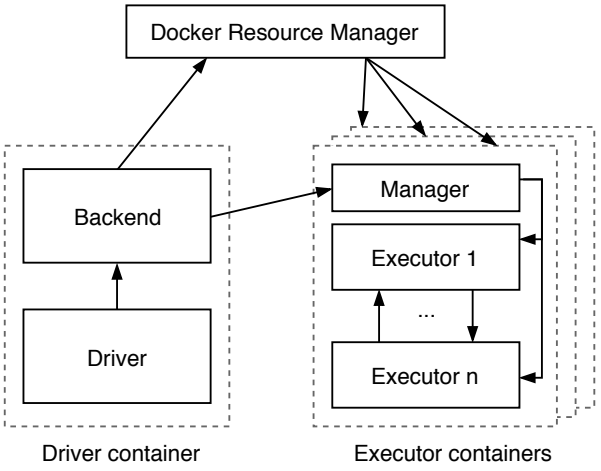



**Figure 1:** Scheme of the Ignis architecture.

Big data processing engines such as Spark have been successfully integrated in a Docker environment [17, 32], showing that performance differences between non-containerized and containerized versions are small. However, while Docker is widely used in the industry, its adoption in the HPC world is not very common. There are important efforts in the research community to deal with some of its limitations. For example, authors in [2] developed a secure way of running Docker containers on a HPC environment avoiding the privilege scalation problems. To improve the bandwidth and throughput of HPC jobs when using containers, other researchers [7] propose a method to allow Docker containers to take advantage of a Infiniband network when running HPC applications. Finally, in a recent work [29], the authors demonstrate how Docker containers can be integrated with HPC environments and run MPI applications with cloud-enabled schedulers.

## 3. Architecture of the Ignis Framework

Ignis is divided into four independent main modules which run inside Docker containers: BACKEND, MANAGER, DRIVER and EXECUTOR (see Figure 1). They are coded in different languages, using Apache Thrift for the inter-module communications. We can summarize the interactions and main goals of the different Ignis modules as follows. The DOCKER RESOURCE MANAGER is responsible of launching and destroying containers and also of assigning the required resources to them. Since it can be used outside of the context of Ignis, the interaction is performed through an HTTP API instead. The DRIVER and the BACKEND share the same container. Users access all the available features of Ignis through a user API defined by the DRIVER. Note that this module is only an interface to the BACKEND, where services that define the logic of the API operations are specified. The BACKEND is responsible of interacting with the DOCKER RESOURCE MANAGER to build a cluster of containers according to the

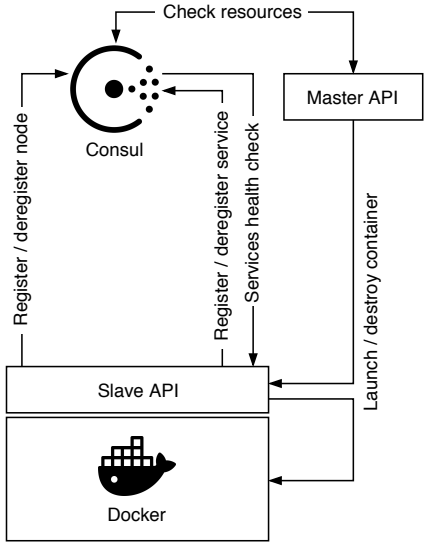



**Figure 2:** DOCKER RESOURCE MANAGER architecture.

```
{
        "image": "test",
        "resources": {
                "cores": 2,
                "memory": "2GB",
                "swap": "0",
                "volumes": [{
                        "size": "50MiB",
                        "mode": "rw-ro",
                        "type": "tmpfs"
                }, {
                        "mode": "rw-rw",
                        "type": "glusterfs"
                }],
                "devices": [],
                # RPC and data ports
                "ports": [2013, 1963]
        },
        "opts": {
                "prefered_hosts": ["node1", "node2"],
                "swappiness": 0
        },
        "events": {
                "on_exit": {
                        "restart": false,
                        "destroy": true
                }
        },
        "args": ["manager", "2013", "6000"]
```

**Figure 3:** Example of a DOCKER RESOURCE MANAGER request.

instructions specified in the driver user code. This module also handles the distribution and exchange of data among executors through the MANAGER. In case some data is lost due to a failure of a cluster node or some of the executors, the BACKEND is able to recompute the corresponding portion of data without requiring a costly replication. Finally, the EXECUTOR MODULE implements for the supported programming languages the operations defined by the BACKEND.

Next, a more detailed description of each module is provided.

### 3.1. Docker Resource Manager

The DOCKER RESOURCE MANAGER executes itself inside Docker containers and is composed of two types of instances: masters and slaves. Master instances manage the available resources in a cluster and they are responsible of launching the containers with the assigned resources through the slave interfaces. Slaves expose the resources of their host when they are deployed. Both client requests and internal calls from masters to slaves are done through HTTP APIs.

The DOCKER RESOURCE MANAGER uses Consul[1], which is a distributed, highly available system that provides a framework for discovering and configuring services within a cluster. Among its main functionalities are service discovery (find new providers of a given service), health checking (check the status of the registered services) and hierarchical key/-value storage. The basic communications between Consul, masters and slaves are illustrated in Figure 2.

When a slave is initialized, the configured resources of that machine are registered in the key-value store provided by Consul. The resources are defined in a granular way so a client can request different types of devices of the same family (e.g., hard disks, GPUs) and even choose a particular device. One important attribute is the *CPU normalizer*, whose goal is to represent the relative performance of a physical/hyperthread core on a cluster of heterogeneous nodes. For instance, a less powerful CPU could specify a normalizer factor of 1.0, whereas another node with a CPU twice as powerful would specify 2.0. In this way, the second node would double the number of virtual cores.

Tasks (in the context of our resource manager) represent containers with assigned resources. They can be included into an existing task group. Otherwise, a new task group would be automatically created for the new task. This feature allows the user to take actions on all the tasks under the same group at the same time. When launching a task, the following parameters can be provided: number of virtual cores, memory, Docker image, arguments for the image, on-exit behaviors, ports to be opened and volumes. A preference node list can be also supplied in such a way that the first node of the list with enough available resources will be selected to execute the container on. Otherwise, the selection process will follow a round-robin scheduling. In Consul, tasks are registered as services and a health check endpoint is provided so Consul can be used to observe the status of the existing tasks. An example of task request is shown in Figure 3. The master is responsible for checking the request and finding a slave node for it. If succeeds, it will interact with the chosen slave node in order to launch the task with the required resources.

Currently, there are three types of supported volumes: local, in-memory and distributed. Local volumes are disk images mounted as loop devices and stored in local disks. When volumes are not in use, they are compressed. In-memory volumes are local volumes whose content is copied to a *tmpfs*, so memory is used instead of disk during the execution of a task. Finally, distributed volumes are symbolic links to directories within a distributed filesystem such as GlusterFS [13].

[1] https://www.consul.io

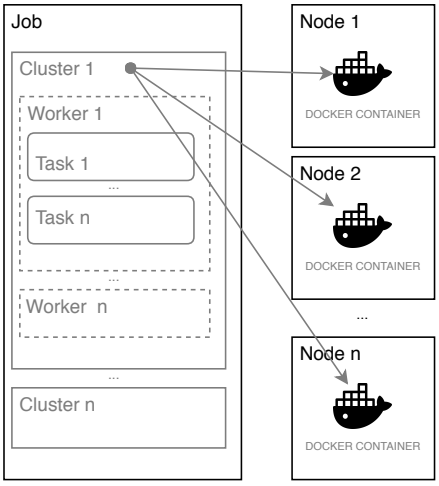



**Figure 4:** Job hierarchy in the Ignis framework.

Volumes can be created with custom permissions (ro, rw) for the first task that mounts it and, optionally, for the same volume group. Group-shared volumes will be available for all the tasks with mounted volumes in the same group. Local volumes will only be available for the group if tasks are in the same host.

All the containers receive the IP address of their host. The requested ports are mapped to random available ports in the host machine.

### 3.2. Driver Module

The DRIVER MODULE is a user API through which users can access all the available functionalities of the Ignis framework. Driver code can be programmed in any of the supported languages (Java, Python and C/C++). This module does not perform any heavy computation and uses Thrift RPC to delegate its work to the BACKEND MODULE. The DRIVER MODULE was designed as a mere interface so the logic has not to be reimplemented for every programming language.

Figure 4 shows the hierarchy of the components of a job in Ignis. A *cluster* is a group of Docker containers distributed in multiple computing nodes. In order to build multi-language applications, at least one *worker* has to be created for each programming language. Each worker contains *tasks*, which are the parallel operations that compose a job. Tasks are instantiated as executor processes inside Docker containers.

Figure 5 shows an example of a driver implemented in two different programming languages, Python and C++, for the well-known Wordcount application. Note that both drivers show a similar behavior and syntax. After initializing the Ignis framework, a cluster of Docker containers is configured and built (lines 4-10, driver code). Several parameters such as Docker image, number of containers, number of cores and memory per container are established. Wordcount has two phases. The first stage consists of a map operation that takes as input a text file and is tokenized into key-value pairs $(word, 1)$. This task in the example of Figure 5 was implemented in Python. As a consequence it is necessary to create a Python worker in the cluster (line 13, driver code). Note that the worker is mandatory and it is not related to the programming language in which the driver code is written. Once the worker is created, the map task is defined indicating the file and class paths where the function to be applied can be found (line 16, driver code). We must highlight that for languages that support source code serialization as Python or Java, references to functions can be used instead. Resulting key-value pairs are stored in *words*.

In the following phase of the Wordcount job all the keys are grouped together and the values for similar keys are added up to find the occurrences for a particular word. To illustrate the multi-language support of Ignis that reduction task in the example was implemented in C++. As a consequence, it is necessary to create a C++ worker (line 19, driver code). Data can be shared between different workers using the `importData` function (line 22, driver code). It is worth noting that this function is only necessary when a task requires data generated by a previous one that belongs to a different worker. The driver code ends storing the final results in a file.

Lazy evaluation is performed so tasks in the driver code are only executed when a result is required explicitly. In Figure 5 the trigger that causes the tasks to be launched is writing the final result to file (line 26, driver code). This approach is also followed by Spark where RDDs are computed lazily the first time they are used in an action [34].

### 3.3. Backend Module

The BACKEND MODULE and the DRIVER MODULE are executed in the same Docker container (see Figure 1). The BACKEND MODULE has the services which defines the logic of the DRIVER MODULE. For instance, `reduceByKey` consists of three steps: searching and grouping the keys, and accumulating values with the same key. These operations are defined in the BACKEND MODULE, but their implementations for a particular programming language are done in the EXECUTOR MODULE.

The BACKEND MODULE is also in charge of making the requests to the DOCKER RESOURCE MANAGER with the aim of building the cluster following the properties specified in the driver code. Tasks are stored by the BACKEND MODULE, which are instantiated as executor processes inside the cluster containers. Therefore, each task uses several executors to apply in parallel the same operation to multiple data items. Performance optimizations can be done such as placing executors with high interaction together in the same host.

The distribution and exchange of data among executors is also handled by this module. Data exchange is an asynchronous operation required by functions such as `sort`, `shuffle` or `reduce`. The BACKEND MODULE determines the data to be sent, and the addresses of the source and destination executors for each communication. Once the exchange service is initiated all the messages are sent from the executor processes following the directives of the BACKEND MODULE. The service is the responsible of building connections among executors and allocating a buffer to write and read

```python
# Initialization of the framework
ignis.Ignis.start()
# Resources/Configuration of the cluster
prop = ignis.IProperties()
prop["ignis.executor.image"] = "wordcount"
prop["ignis.executor.instances"] = "2"
prop["ignis.executor.cores"] = "4"
prop["ignis.executor.memory"] = "2GB"
# Construction of the cluster
cluster = ignis.ICluster(prop)

# Initialization of a Python Worker in the cluster
worker_python = ignis.IWorker(cluster, "python")
# Task 1 - Python: tokenize text into pairs ('word', 1)
text = worker_python.readFile("text.txt")
words = text.flatMap("wordcount/wd.py:Split")

# Initialization of a C++ Worker in the cluster
worker_cpp = ignis.IWorker(cluster, "cpp")
# Task 2 - C++: reduce pairs with same word and obtain totals
# Transfer data from Task 1 - Python
words_cpp = worker_cpp.importData(words)
count = words_cpp.reduceBykey("wordcount/libwd.so:Join")

# Print results to file
count.saveAsFile("wordcount.txt")

# Stop the framework
ignis.Ignis.stop()
```

```cpp
// Initialization of the framework
Ignis::start();
// Resources/Configuration of the cluster
prop = std::make_shared<IProperties>();
(*prop)["ignis.executor.image"] = "wordcount";
(*prop)["ignis.executor.instances"] = "2";
(*prop)["ignis.executor.cores"] = "4";
(*prop)["ignis.executor.memory"] = "2GB";
// Construction of the cluster
cluster = make_shared<ICluster>(prop);

// Initialization of a Python Worker in the cluster
worker_python = make_shared<IWorker>(cluster, "python");
// Task 1 - Python: tokenize text into pairs ('word', 1)
text = worker_python.readFile("text.txt");
words = text.flatMap("wordcount/wd.py:Split");

// Initialization of a C++ Worker in the cluster
worker_cpp = make_shared<IWorker>(cluster, "cpp");
// Task 2 - C++: reduce pairs with same word and obtain totals
// Transfer data from Task 1 - Python
words_cpp = worker_cpp.importData(words);
count = words_cpp.reduceBykey("wordcount/libwd.so:Join");

// Print results to file
count.saveAsFile("wordcount.txt");

// Stop the framework
Ignis::stop();
```

**Figure 5:** Wordcount driver code example using Python (left) and its equivalent C++ code (right).

data in the connections. Once all the messages are sent, the exchange service stops. Note that for efficiency reasons executors in the same host exchange data through the shared memory.

Finally, we must highlight that Ignis is able to recover after a failure of a cluster node or some of the executors. The BACKEND MODULE is able to follow the task trace of the affected executors in such a way that only those executors are reallocated and recomputed. It means that even if the input data of an executor is lost, only the tasks necessary to recompute that portion of data are performed. This process is done without the intervention of the user. An experimental evaluation of the fault tolerance mechanisms of Ignis is shown in Section 6.

### 3.4. Manager Module

The MANAGER MODULE is the connection point between the BACKEND MODULE and the executor processes (see Figure 1). Any command from the BACKEND MODULE to the executor processes goes through this module, which includes launching the executors in the containers. In the scenario of no answer from an executor process, the MANAGER MODULE kills the corresponding executor and informs the BACKEND MODULE in order to start the recovery process.

### 3.5. Executor Module

The operations defined in the BACKEND MODULE are implemented for each supported programming language in the EXECUTOR MODULE. Therefore, adding a new language to Ignis only requires to implement those operations in the corresponding language.

Most of the driver functions such as `map` or `reduceByKey` are meta-functions. That is, generic functions that require another function to perform an internal operation. For example, `flatMap` in the codes of Figure 5 apply the `Split` function to the input text (line 16, driver code). Those functions in Ignis are defined using a common interface based on the number of input and output parameters. Figure 6 shows as example the `Split` and `Join` functions of the Wordcount application in Python and C++. `Split` tokenizes a text into key-value pairs (*word*, 1). Therefore, it is a typical flat operation that produces an arbitrary number (zero or more) values for each input value. The difference with a `map` operation is that it must produce one output value for each input. In the Ignis API, *IFlatFunction* represents a flat operation, where the *call* method always contains the implementation of the function.

`Join` takes two count values for the same word and accumulates them. Consequently, multiple values are reduced iteratively to a single value. In this case, *IFunction2* represents an operation that takes two arguments and generates only one result. Note that in the C++ code data types of the input and output parameters should be specified.

The context object is also passed as argument to `Split` and `Join` (Python - lines 4 and 10, C++ - lines 4 and 19), which allows the user to access more information such as the properties defined in the driver code.

On the other hand, some applications may need to perform certain specific tasks for each executor process. For instance, opening a database connection, reading a particular file or preparing the environment before processing. To facilitate this task all the interfaces to the functions in the Ignis

```python
# Tokenize text into pairs ('word', 1)
class Split(IFlatFunction):

  def call(self, elem, context):
    return [(word, 1) for word in elem.split()]

# Reduce pairs with same word and obtain totals
class Join(IFunction2):

  def call(self, elem1, elem2, context):
    return elem1 + elem2
```

```cpp
// Tokenize text into pairs ('word', 1)
class Split : public api::function::IFlatFunction<string, pair<string,int64_t>>
{
   api::Iterable<pair<string,int64_t>> call(string& line, api::IContext& context) {
      // Vector to accumulate pairs
      vector<pair<string,int64_t>> pairs;
      // String stream tokenizer
      stringstream words(line);
      // Iterator
      istream_iterator<string> begin(words), end;

      for(it = begin; it != end; it++) pairs.emplace_back(*it, 1);
      return(pairs); }
};

// Reduce pairs with same word and obtain totals
class Join : public api::function::IFunction2<int64_t, int64_t, int64_t>
  {
  size_t call(int64_t& count1, int64_t& count2, api::IContext& context) {
      return count1 + count2; }
};
```

**Figure 6:** `Split` and `Join` functions for the Wordcount application using Python (left) and its equivalent C++ code (right).

API provide a *before* method, which is called at the beginning of the executor, and an *after* method, which is called once at the end of the processing. This functionality is similar to the *setup* and *cleanup* methods in Hadoop, but it is not supported by Spark.

## 4. Data Storage

Ignis provides several options for data storage thanks to its common storage interface. In this way, it is possible to create different representations of the data without modifying the operations. Users can choose the type of storage that best fits their application taking into account the limitations of their particular execution environment. Only it is necessary to specify the selection as a property in the driver code. Next we explain the different storage options supported by Ignis:

- *In-Memory:* This is the Ignis default option and provides the fastest performance since all data is stored in memory.
- *Raw memory storage:* A serialized representation in memory that allows to remove extra space introduced by objects at the expense of an additional overhead. Serialization protocols are the same used for data exchange between executors. The memory buffer is compressed by Zlib [16], which has 9 compression levels that can be changed when defining the properties in the driver code. By default, level 6 is applied. The disadvantage of serializing data and applying compression is that elements are not indexed and must be accessed sequentially.
- *Index Raw memory storage:* This is a special Python storage method to overcome the Global Interpreter Lock (GIL) limitations, which causes only one thread to execute at a time [26]. This type of storage uses a binary representation, uncompressed, with a table to index the data elements. Data and the index table are stored in shared memory, so they can be accessed by multiple Python processes. This storage replaces the default in-memory storage when using a Python worker and the number of cores per container is greater than one (see the Wordcount driver code of Figure 5).
- *Raw Disk storage:* This is a variation of the raw memory method where the buffer uses a file mapping approach. In this way, while a portion of the data is in memory, the remaining is stored in disk. This storage option allows to work with large amounts of data that can not be completely stored in memory.

Data in Ignis is ephemeral. It means that when data previously computed is used as input of a new task, it is discarded after use. But, what happens when multiple tasks have the same input data?. In this case, the best solution in terms of efficiency is not discarding the data to avoid extra computations for each task. To deal with this issue, Ignis provides two mechanisms to alter the persistence of data: `cache` and `persist`. In particular, the `cache` method hints that data should be kept in memory after the first time is computed, because it will be reused. The `persist` method allows users to choose a different storage option for the data: in-memory, raw memory storage or raw disk storage. When data is no longer needed, it must be explicitly removed by users with `uncache` or `unpersist` indistinctly.

## 5. Ignis API

To use Ignis, developers should write a driver program that implements their application at high-level (see the example of Figure 5). As we explained in Section 3.5, some of the driver functions such as `map` require another one to execute. In that case it is also necessary to implement those

functions. We must highlight that although the Ignis code uses a sequential notation, operations on data are performed in parallel. In order to facilitate the adoption from the Big Data community, the Ignis API was inspired by the Spark API in such a way that Ignis codes are easily understandable by users who are familiar with Spark. Next we provide details about the current functions supported by Ignis:

- *Managing files:* Ignis is able to load and save data from/to any location in a (distributed) file system. Currently data can only be read from text files using `readFile`. Data can be saved to a file in plain text (`saveAsFile`) or JSON format (`saveAsJSON`). It is possible to save distributed data into one single file or several files (one file per executor that owns a portion of the data).

- *Map functions:* The common characteristic to functions belonging to this type is that they apply the same function to each element in the data. As a result of the transformation, the output could be of different size with respect to the input. The available functions are: `map`, `flatMap`, `filter`, `keyBy` and `values`. The first two are very similar. While `map` applies a one-to-one transformation to the input data, `flatMap` generates an arbitrary number of results. `filter` is a map operation that pick elements from the input data matching a predicate. `keyBy` returns a $(key, value)$ pair after applying a function to the *value* argument with the aim of calculating the $key$. `values` does not need an extra function, it only returns *value* from a $(key, value)$ pair.

- *Reduce functions:* The `reduction` method aggregates all the elements in the input data using a function. `aggregation` is a sort of reduction where the type of the input and output data is different. Two functions are necessary, the first one is applied to each element in a data partition, and the second one combines the partial results obtained for each partition. `reduceByKey` and `aggregateByKey` are variations where the operation is performed only among elements with the same key in such a way that the final result is a set of unique pairs with values calculated using `reduce` or `aggregate` operations, respectively.

- *Sort functions:* In order to sort elements Ignis provides two functions: `sort` and `sortBy`. The first method uses the natural order and does not need any additional function. `sortBy` allows to use a user-defined function to specify the order of the elements. If the result of applying that function to two elements is *true*, then the first element should precede the second one. Both methods support ascending and descending order.

- *Shuffle functions:* The `shuffle` method balances the number of elements to be processed by each executor, keeping the same order. This operation is useful to preserve performance after an operation that greatly unbalances the data. It should be invoked by the user. The function `importData`, which allows different workers to share data, performs an internal shuffle operation if the number of executors for each worker is different.

- *Other functions:* Ignis implements several operations that return a value to the driver code, but they do not modify or generate new stored data. Spark refers to this type of operations as *actions*. In particular, Ignis supports `count`, `take`, `takeSample` and `collect`. The most basic operation is `count` that returns the number of elements of a stored data collection. `collect` returns a collection with all the elements stored in the executors of a task. `take` applies a `collect` operation but obtains only the first $n$ elements, where $n$ is chosen by the user. `takeSample` returns a random sample of $n$ elements from the distributed data, with or without replacement. Finally, `parallelize` distributes the elements of a collection among the executors to form a distributed dataset. In this case new stored data is created.

  As we mentioned, the above functions exchange data with the driver. With the aim of improving overall performance, Ignis also implements optimized versions for scenarios where data to be exchanged is of small size.

## 6. Experimental Results

In this section we evaluate Ignis using several applications in terms of performance, scalability and fault tolerance. A comparison with Spark is also provided.

### 6.1. Hardware Platform and Software

The experiments shown in this section were carried out on an 8-node cluster, where each node consists of:

- CPU: 2 x Intel Xeon E5-2630v4 (2.2Ghz, 10 cores)
- Memory: 384 GB of RAM
- Storage: 8 x 4TB 7.2k SATA
- Network: 2 x 10GbE

It is a Linux cluster running CentOS 7 (kernel 3.10.0), Docker 18.09.1-ce and Spark 2.2.0 (with YARN [30] as cluster manager). GlusterFS on XFS was used by Ignis as distributed file system.

In order to illustrate the benefits of our proposal we have considered four applications: *Minebench*[2], *K-Means*, *Sort* and *Conjugate Gradient*. The first three represent different types of application patterns for which Spark is considered the best performing Big Data framework with respect to other approaches such as Hadoop. For instance, *Minebench* can be considered a chain of `map` operations, while *K-Means* uses an iterative MapReduce model. For completeness we

[2]Do not confuse with the data mining benchmark suite NU-MineBench.

have also analyzed the *Conjugate Gradient*, which is one of the most relevant iterative solvers for systems of linear equations in the HPC world. Using these applications we will compare the performance of Ignis and Spark. On the one hand, we will demonstrate that, while Python is natively supported by Ignis, an important overhead is caused by using system pipes for data transferring among executors in Spark. As a consequence, Spark degrades both performance and scalability since Python is treated as an external language. On the other hand, we will show the benefits of our approach when running applications coded using several programming languages.

Instead of considering only speedup to assess the scalability, we will consider a better alternative based on plotting the raw execution time when using different number of cores on a cluster [14]:

- Strong scaling: for a fixed problem, a straight line with slope -1 indicates good scalability, whereas any upward curvature away from that line indicates limited scalability.
- Weak scaling: for a sequence of problems with a fixed amount of work per core, a horizontal straight line indicates good scalability, whereas any upward trend of that line indicates limited scalability.

### 6.2. Minebench

Minebench[3] performs the calculation of SHA-256 hashes imitating the Proof-of-Work algorithm used in the Bitcoin protocol [25]. This algorithm has two phases which are implemented using two chained `map` operations. The first map is data-intensive, while the second is a compute-intensive task. In particular, in the first stage a set of Bitcoin transactions are grouped together forming a block proposal. A binary Merkle tree [23] is calculated for those transactions and its Merkle root hash is added to a block header in addition to other attributes such as protocol version, hash of the previous block header, the current timestamp and the difficulty, through which is calculated the network target. The network target determines the threshold under which the hash of the proposed block header must be considered valid. The second stage calculates the hash of the block header iteratively while the condition is not met. In order to obtain different results, another attribute is present in the block header: the *nonce*. It is an integer that is changed in every iteration so the resulting hash also changes with it.

Two different implementations of the application were considered in the tests. The first one was programmed using only Python. In the second version, two different programming languages were used: Python and C++. In this case, the first phase of the application uses Python since it requires handling data, while the compute-intensive tasks use C++ to achieve the best results in terms of performance.

Figure 7 shows the scalability results obtained by Ignis and Spark when running the Proof-of-Work algorithm using both implementations (i.e., only Python and Python & C++) on our cluster. The strong scaling tests were obtained using a 120MB input file containing 300k blocks, while the weak scaling experiments start from a 120MB input file (one core) to reach 4.2GB and 9,600k blocks (32 cores). A least-squares fit is also provided to estimate the scalability results using up to 256 cores. In addition, graphs show a line corresponding to the ideal scalability.

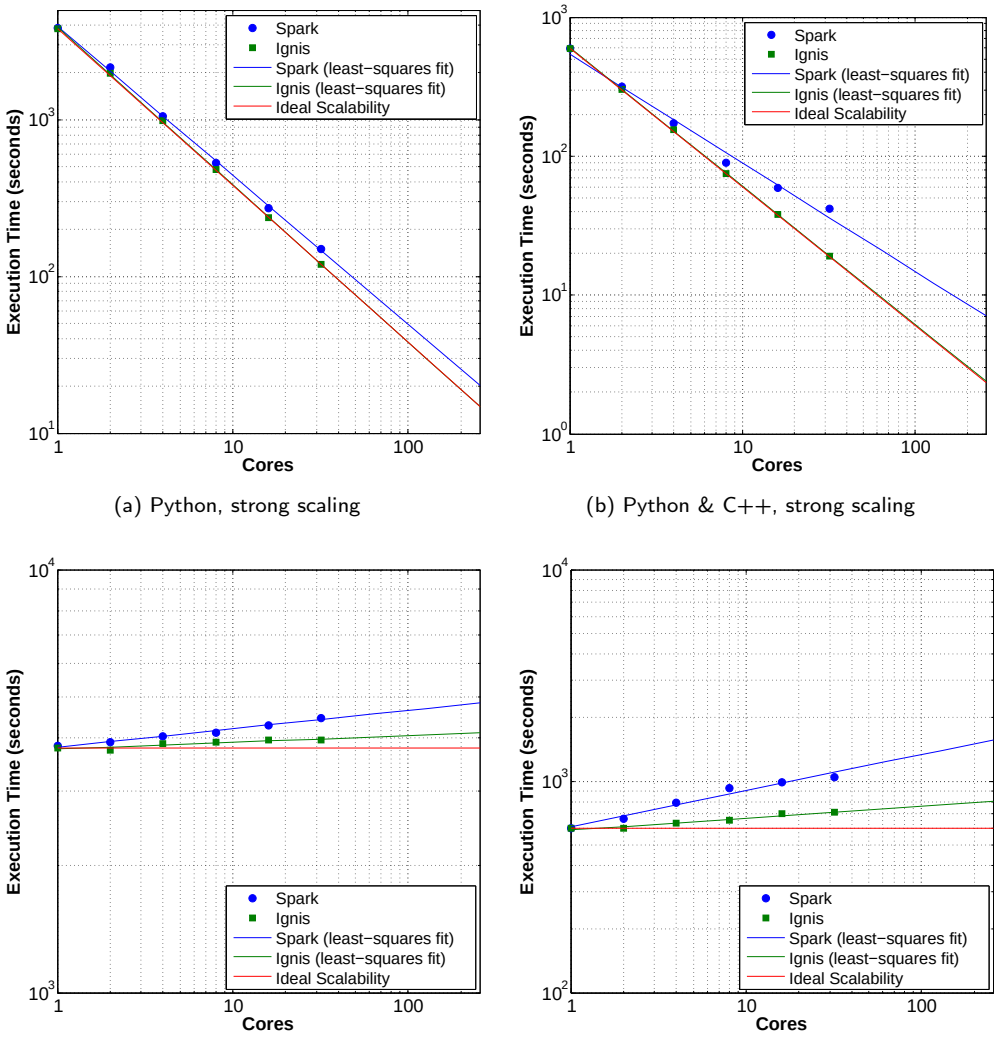
First, we analyze the strong scaling results. When considering the Python application (Figure 7(a)), Spark and Ignis obtain similar results with a small number of cores. However, as the number of executors increases, the cost of starting JVMs and transferring data through system pipes to the Python processes degrades the Spark global performance. Therefore, running Python applications on Spark prevents from reaching high levels of parallelism. For example, Ignis is 1.3× faster than Spark using 32 cores. This behavior is even more clear running the multi-language application (Figure 7(b)). Since Spark sends data from Python to C++ processes through the JVM, the number of pipe operations increases. In this way, the overhead is greater than the one observed for the Python implementation. In this case, Ignis is 2.2× faster than Spark using 32 cores. The least-square fits point out that execution times diverge and, as a consequence, the performance difference between Ignis and Spark will increase when considering more cores. We can conclude that our framework shows a very good behavior in terms of strong scalability, always close to the ideal case (red line). On the other hand, the multi-language implementation clearly outperforms the Python benchmark.

Weak scaling results are displayed in Figures 7(c) and 7(d). In those tests we keep constant the amount of work performed for each core. The overhead detected in the results considering the Python code demonstrates that Python is not natively supported in Spark like Java or Scala. Note how the Spark scalability is getting away from the ideal case (horizontal red line) as the number of cores increases. It can be observed that Ignis shows a better behavior. When considering the multi-language code, the overhead of Spark becomes even more noticeable. That overhead is due to the exchange of data between processes. Pipes use the hard disk for those exchanges, causing an important bottleneck in the performance. However, Ignis still keeps a good scalability.

### 6.3. K-means

*K-Means* is a classical machine learning algorithm for data clustering, and it is a good example of an iterative MapReduce application pattern. The goal of this algorithm is to classify a given data set through a certain number of clusters ($K$ clusters). Each cluster has a centroid. The algorithm works iteratively in such a way that every iteration each data point is assigned to the nearest centroid, and the $K$ centroids are recalculated as barycenters of the clusters resulting from the previous assignment step. These operations are compute-intensive tasks. The algorithm iterates until the centroids do not change their location or other criterion is fulfilled. The final goal of the algorithm is that points within the same cluster are as similar as possible (i.e., high

[3] Publicly available at: https://github.com/brunneis/minebench




(a) Python, strong scaling

(b) Python & C++, strong scaling

(c) Python, weak scaling

(d) Python & C++, weak scaling

**Figure 7:** Study of the scalability of Ignis and Apache Spark running the Minebench application. Axis are in log scale.

intra-class similarity), while points from different clusters are as dissimilar as possible (i.e., low inter-class similarity).

Spark provides its own implementation of *K-means* in MLlib (Machine Learning Library) [22]. We have implemented a pure Python version for Spark and Ignis of the algorithm described in [3]. A Python-C++ multi-language version of that algorithm was also executed on the Ignis framework. In that version compute-intensive tasks were implemented in C++, while the *K-means* algorithm was specified in Python. For fair comparison we also included the Spark performance results for the MLlib version. Experiments were conducted using the NUS-WIDE dataset [6], which contains 269,648 images with 500 attributes per image.

Figure 8 shows the strong scalability of the *K-Means* application. Results were obtained after 10 iterations using different number of clusters: $K = 12$ (top graph) and $K = 81$ (bottom graph). As we noted above, Spark (Python) and Ignis (Python) execute the same *K-means* code on both platforms. Tests point out that the continuous exchange of data between the Python processes and the JVM degrades the Spark performance when running an iterative application like *K-Means*. Performance differences between Ignis and Python grow as the parallelism increases. On the other hand, the Spark implementation that uses MLlib outperforms Spark (Python) and Ignis (Python) even though the driver code was also programmed in Python. This behavior indicates that the `KMeans` method in a Python Spark driver code is just a wrapper of the most efficient Scala implementation of the algorithm. In this way, the overhead caused by the pipe operations disappears. In any case, the Python-C++ multi-language code on Ignis is the fastest implementation.

To summarize the benefits of using Ignis with respect to Spark when running iterative MapReduce applications we show in Figure 9 the previous *K-Means* results expressed in terms of speedup. The top graph illustrates how Python is considered a non-native language by Spark. To do so, the speedup between the pure Python *K-Means* versions run-

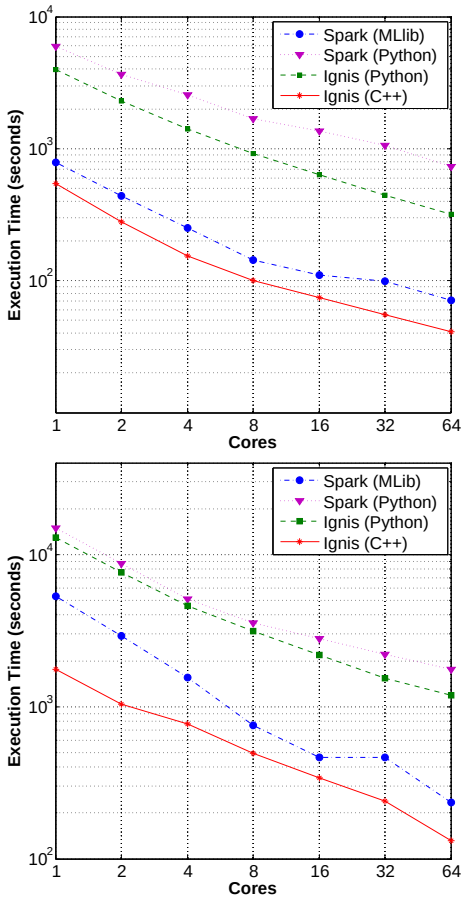



**Figure 8:** Study of the scalability of Ignis and Apache Spark running the K-Means application: $K = 12$ (top) and $K = 81$ (bottom). Axis are in log scale.

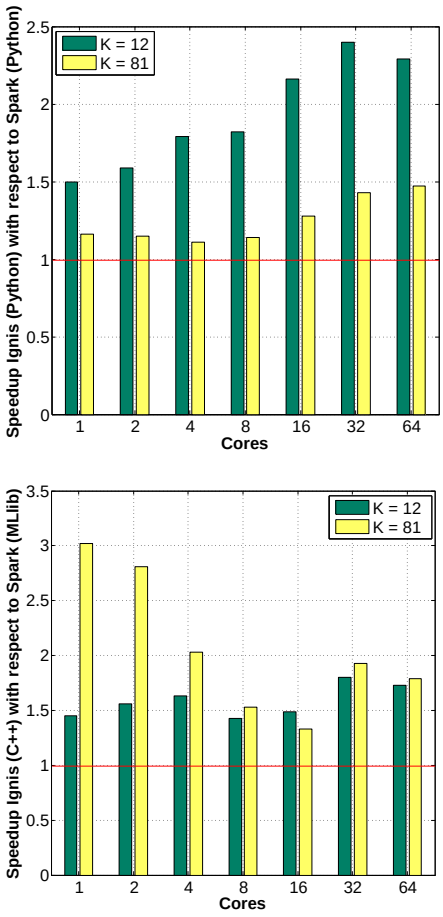



**Figure 9:** Speedup of Ignis with respect to Apache Spark running the K-Means application.

ning on Spark and Ignis for each number of cores is calculated. According to the results, Ignis is on average 1.94× and 1.25× faster than Spark with $K = 12$ and $K = 81$, respectively. Speedups up to 2.4× were reached. The bottom graph displays the speedup between the best performing Spark version (MLlib) and the Ignis multi-language code for a particular number of cores. In this case, Ignis is on average 1.58× and 2.06× faster than Spark with $K = 12$ and $K = 81$, respectively. Therefore, Ignis is able to handle efficiently the combination of different programming languages in one iterative MapReduce application, extracting the maximum performance from the considered parallel system.

On the other hand, one of the main features of Ignis is that it provides fault tolerance efficiently. As we pointed out in Section 3.3, if some data is lost, Ignis has enough information about how it was derived. In this way, only those operations needed to recompute the corresponding portion of data are performed. Therefore, lost data can be recovered without requiring costly replication. Next we will evaluate the cost of reconstructing a data partition after a node failure in the *K-Means* application.

Figure 10 compares the running times for 15 iterations of *K-Means*, with one where a node fails at the start of the 10th iteration. We also included the Spark times for the same scenario. Tests were performed considering the Python implementation using 16 cores on a 4 nodes cluster. Until the end of the 9th iteration, the Ignis iteration times were about 3 seconds. In the 10th iteration, one of the nodes is killed, resulting in the loss of the tasks running on that machine and the data partitions stored there. As a consequence, Ignis reruns these tasks on other machine nodes, rebuilding the data, which increases the iteration time to 15s. Once the lost data is reconstructed, the Ignis iteration time goes back down to 3s. Spark behaves similarly, increasing the 10th iteration time from about 7s to 19s, going back to the normal iteration time afterwards.

Note that with a checkpoint-based fault recovery mechanism, recovery would likely require rerunning at least several iterations, depending on the frequency of checkpoints.

## 6.4. Sort Benchmark

Sorting is a very common and useful data-intensive operation, thus it is supported by Spark and Ignis. Elements in Ignis are sorted by means of the MergeSort algorithm where elements are distributed by a regular sampling among the ex-

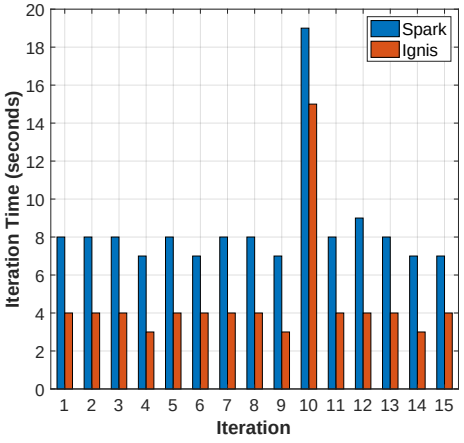



**Figure 10:** Iteration times for K-Means in presence of a failure. One node failed at the start of the 10th iteration.

ecutors [18]. This task requires that executors exchange data. In both platforms the user can define a comparison function to be used together with `sort`, but the algorithm itself can not be modified. In contrast to the `map` operation, Spark does not allow to implement that user-defined function in a programming language different than the one used in the driver. That limitation does not appear in Ignis since any combination of programming languages for the driver and tasks is permitted. In this way, a Ignis driver written in Python could use a Java or C++ comparison function.

Performance tests were carried out sorting in ascending order 35GB of text data, which contains about two billion lines. Each line has from 10 to 30 bytes of random text. We have compared the `sort` built-in capability of Spark with Ignis. Two different implementations in Ignis were analyzed. In the first one, the code was completely programmed in Python, while the second has a Python driver and the `sort` operation uses a user-defined C++ function for comparison purposes. Results are displayed in Figure 11. The top graph shows the execution times up to 64 cores. We can observe that while the behavior of the strong scalability is similar for all the cases, Ignis outperforms Spark in terms of execution time even for the Python implementation. Using a Python-C++ multi-language application is again the best option. Speedups, which are displayed in the bottom graph, were calculated using as reference the Spark execution with one core. Results confirm the previous observations regarding computing times. For instance, Ignis sorts data 1.14× and 1.72× faster than Spark using 64 cores when considering the Python and the multi-language versions, respectively.

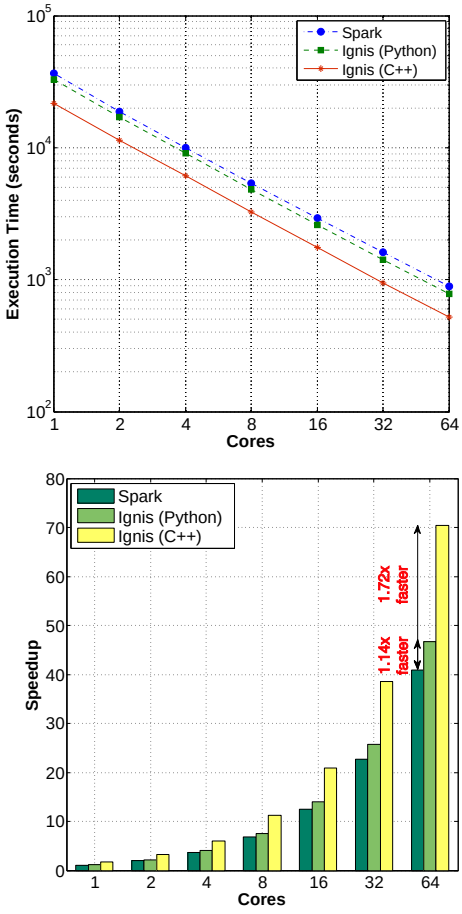



**Figure 11:** Study of the scalability of Ignis and Apache Spark sorting 35GB of text data.

## 6.5. Conjugate Gradient (CG)

The *Conjugate Gradient* (CG) is an iterative method for solving sparse, large systems of linear equations. It is considered one of the most important computational kernels lying at the heart of many HPC scientific and engineering applications. A CG iteration involves the following compute-intensive linear algebra operations [4]: one sparse matrix-vector product, three vector updates, and two inner products.

We have implemented the CG method for Spark and Ignis using Python. BLAS routines were carried out using the Intel MKL library. Our tests were conducted considering as input a $100k \times 100k$ sparse matrix with 900M nonzeros. The method was executed until 100 iterations were reached. The experimental results are shown in Figure 12. The top graph illustrates the strong scalability of CG using up to 64 cores. Although the scalability results are not very good, Ignis has been proven to be clearly faster than Spark running this typical HPC application. For example, as it is displayed in the figure at the bottom, Ignis is 1.43× faster than Spark with 64 cores. In that graph speedups were computed taking as reference the Spark execution time with one core.

## 7. Conclusions

In this work we introduced Ignis, a new Big Data framework whose main contribution is twofold. First, unlike standard Big Data engines such as Hadoop and Spark, the new system allows to execute natively applications implemented using non-JVM languages. Second, it facilitates the development of multi-language applications in such a way that

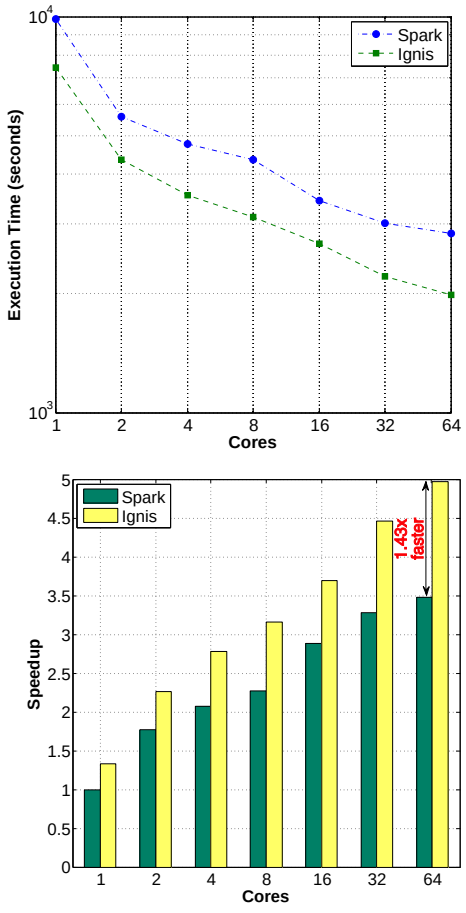



**Figure 12:** Study of the scalability of Ignis and Apache Spark running the CG application. Axis are in log scale.

different computing tasks could be implemented in the programming language that best fits each of them. In this way, it is possible to exploit efficiently the capabilities of different programming languages in the same application. The system runs completely inside Docker containers isolating the execution environment from the physical machine.

The experimental evaluation was carried out using four different applications with the aim of covering some of the most representative algorithmic patterns in Big Data and scientific computing. First, an application that performs the Proof-of-Work algorithm used in the Bitcoin protocol, which corresponds to chain several map operations. Second, an iterative MapReduce algorithm such as K-Means. Third, a Sort operation, which is one of the most important kernels at the core of many other parallel applications. And finally, the Conjugate Gradient, a very well-known HPC method to deal with systems of linear equations. Spark and Ignis frameworks have been compared. According to the results several global conclusions can be made. Spark does not natively support Python applications since an important degradation in performance and scalability was observed. We find the cause in the exchange of data between Python processes and the JVM through system pipes. These pipe operations were completely removed in Ignis, which clearly outperforms Spark when running Python applications. For instance, running K-Means is up to 2.4× faster than Spark. A slowdown in the Spark performance was also detected when running multi-language applications due to the same causes. On the contrary, Ignis allows to execute efficiently an application written in several programming languages without additional overhead. In this way, it is possible, for example, use Python for handling data while in the same application C++ code deals with compute-intensive tasks. Finally, in addition to Java and Python, Ignis currently supports C/C++, which is probably the most important programming language in HPC. As a consequence, Ignis allows to execute applications belonging to HPC and Big Data worlds in the same framework. Therefore, Ignis means a new step forward towards the real convergence of HPC and Big Data.

## References


[1] Apache Thrift, . https://thrift.apache.org/. [Online; accessed April, 2019].

[2] Azab, A., 2017. Enabling Docker Containers for High-Performance and Many-Task Computing, in: IEEE Int. Conference on Cloud Engineering (IC2E), pp. 279–285.

[3] Bahmani, B., Moseley, B., Vattani, A., Kumar, R., Vassilvitskii, S., 2012. Scalable K-means++. Proceedings of the VLDB Endowment 5, 622–633.

[4] Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., van der Vorst, H., 1994. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. Society for Industrial and Applied Mathematics.

[5] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K., 2015. Apache Flink: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull. 38, 28–38.

[6] Chua, T.S., Tang, J., Hong, R., Li, H., Luo, Z., Zheng, Y., 2009. NUS-WIDE: A Real-world Web Image Database from National University of Singapore, in: Proc. of the ACM Int. Conference on Image and Video Retrieval (CIVR), pp. 48:1–48:9.

[7] Chung, M.T., Le, A., Quang-Hung, N., Nguyen, D., Thoai, N., 2016. Provision of Docker and InfiniBand in High Performance Computing, in: Int. Conference on Advanced Computing and Applications (ACOMP), pp. 127–134.

[8] Dean, J., Ghemawat, S., 2004. MapReduce: Simplified Data Processing on Large Clusters, in: Symposium on Operating System Design and Implementation, pp. 10–10.

[9] Ding, M., Zheng, L., Lu, Y., Li, L., Guo, S., Guo, M., 2011. More Convenient More Overhead: The Performance Evaluation of Hadoop Streaming, in: Proc. of the ACM Symposium on Research in Applied Computation, pp. 307–313.

[10] Docker, . https://www.docker.com/. [Online; accessed April, 2019].

[11] Ekanayake, S., Kamburugamuve, S., Fox, G.C., 2016. SPIDAL Java: High Performance Data Analytics with Java and MPI on Large Multicore HPC Clusters, in: Proc. of the 24th High Performance Computing Symposium, pp. 3:1–3:8.

[12] Fox, G.C., Qiu, J., Kamburugamuve, S., Jha, S., Luckow, A., 2015. HPC-ABDS High Performance Computing Enhanced Apache Big Data Stack, in: 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 1057–1066.

[13] GlusterFS, . https://www.gluster.org/. [Online; accessed April, 2019].

[14] Heath, M.T., 2015. A tale of two laws. The International Journal of High Performance Computing Applications 29, 320–330.

[15] Jython, . http://www.jython.org/. [Online; accessed April, 2019].

[16] Kukunas, J.T., Gopal, V., Guilford, J., Gulley, S., van de Ven, A., Feghali, W., 2014. High Performance ZLIB Compression on Intel

Architecture Processors. Technical Report. Intel.
[17] Lei, Z., Du, H., Chen, S., Zhu, C., Liu, X., 2016. DCSPARK: Virtualizing Spark using Docker containers, in: Int, Conference on Audio, Language and Image Processing (ICALIP), pp. 13–18.
[18] Li, X., Lu, P., Schaeffer, J., Shillington, J., Wong, P.S., Shi, H., 1993. On the Versatility of Parallel Sorting by Regular Sampling. Parallel Computing 19, 1079–1103.
[19] Lu, X., Islam, N.S., Wasi-Ur-Rahman, M., Jose, J., Subramoni, H., Wang, H., Panda, D.K., 2013. High-Performance Design of Hadoop RPC with RDMA over InfiniBand, in: 42nd Int. Conference on Parallel Processing, pp. 641–650.
[20] Lu, X., Liang, F., Wang, B., Zha, L., Xu, Z., 2014. DataMPI: Extending MPI to Hadoop-Like Big Data Computing, in: IEEE 28th Int. Parallel and Distributed Processing Symposium, pp. 829–838.
[21] Malitsky, N., Chaudhary, A., Jourdain, S., Cowan, M., O'Leary, P., Hanwell, M., Van Dam, K.K., 2017. Building near-real-time processing pipelines with the Spark-MPI platform, in: New York Scientific Data Summit (NYSDS), pp. 1–8.
[22] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., Xin, D., Xin, R., Franklin, M.J., Zadeh, R., Zaharia, M., Talwalkar, A., 2016. MLlib: Machine Learning in Apache Spark. The Journal of Machine Learning Research 17, 1235–1241.
[23] Merkle, R.C., 1980. Protocols for public key cryptosystems, in: IEEE Symposium on Security and Privacy, pp. 122–122.
[24] Misale, C., Drocco, M., Tremblay, G., Martinelli, A.R., Aldinucci, M., 2018. PiCo: High-performance data analytics pipelines in modern C++. Future Generation Computer Systems 87, 392 – 403.
[25] Nakamoto, S., 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. URL: http://bitcoin.org/bitcoin.pdf.
[26] Palach, J., 2014. Parallel Programming with Python. Packt Publishing.
[27] Piñeiro, C., Abuín, J.M., Pichel, J.C., 2017. Perldoop2: A Big Data-Oriented Source-to-Source Perl-Java Compiler, in: Proc. of the IEEE Intl. Conf. on Big Data Intelligence and Computing (DataCom), pp. 933–940.
[28] Saha, B., Shah, H., Seth, S., Vijayaraghavan, G., Murthy, A., Curino, C., 2015. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications, in: Proc. of the ACM SIGMOD Int. Conference on Management of Data, ACM. pp. 1357–1369.
[29] Saha, P., Beltre, A., Govindaraju, M., 2019. Scylla: A mesos framework for container based MPI jobs. CoRR abs/1905.08386. arXiv:1905.08386.
[30] Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., Baldeschwieler, E., 2013. Apache Hadoop YARN: Yet Another Resource Negotiator, in: Proc. of the 4th Annual Symposium on Cloud Computing, ACM. pp. 5:1–5:16.
[31] White, T., 2015. Hadoop: The Definitive Guide. 4th ed., O'Reilly Media, Inc.
[32] Ye, K., Ji, Y., 2017. Performance Tuning and Modeling for Big Data Applications in Docker Containers, in: Int. Conference on Networking, Architecture, and Storage (NAS), pp. 1–6.
[33] Yildiz, O., Zhou, A.C., Ibrahim, S., 2018. Improving the Effectiveness of Burst Buffers for Big Data Processing in HPC Systems with Eley. Future Generation Computer Systems 86, 308 – 318.
[34] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I., 2012. 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, USENIX Association. pp. 2–2.
[35] Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I., 2010. Spark: Cluster Computing with Working Sets, in: Proc. of the 2nd USENIX Conf. on Hot Topics in Cloud Computing (HotCloud), pp. 10–10.