A unified framework to improve the interoperability between HPC and Big Data languages and programming models*

César Piñeiro[a,*], Juan C. Pichel[a]

[a]Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela, 15782 Santiago de Compostela, Spain

Abstract

One of the most important issues in the path to the convergence of HPC and Big Data is caused by the differences in their software stacks. Despite some research efforts, the interoperability between their programming models and languages is still limited. To deal with this problem we introduce a new computing framework called IgnisHPC, whose main objective is to unify the execution of Big Data and HPC workloads in the same framework. IgnisHPC has native support for multi-language applications using JVM and non-JVM-based languages. Since MPI was used as its backbone technology, IgnisHPC takes advantage of many communication models and network architectures. Moreover, MPI applications can be directly executed in an efficient way in the framework. The main consequence is that users could combine in the same multi-language code HPC tasks (using MPI) with Big Data tasks (using MapReduce operations). The experimental evaluation demonstrates the benefits of our proposal in terms of performance and productivity with respect to other frameworks. IgnisHPC is publicly available for the Big Data and HPC research community.

Keywords: Big Data, HPC, MPI, Multi-language, Programming models

1. Introduction

The unification of High Performance Computing (HPC) and Big Data has received increasing attention in the last years. It is a common belief that exascale computing and Big Data are closely associated since HPC requires processing large-scale data from scientific instruments and simulations. But, at the same time, it was observed that tools and cultures of HPC and Big Data communities differ significantly [1]. One of the most important sources of divergence comes from the differences between their software ecosystems. In this way, HPC applications have traditionally been based on MPI (Message Passing Interface) to support inter-node parallel execution, and based on OpenMP or other alternatives to exploit intra-node parallelism. However, Big Data programming models are based on interfaces like Hadoop [2] or Spark [3]. In addition to different programming models, programming languages also differ between both communities: being Fortran and C/C++ the most common languages in HPC applications, and Java, Scala, or Python being the most common languages in Big Data applications.

This divergence between programming models and languages sets out a convergence problem, not only related to the interoperability of the applications but also to the interoperability between data formats from different programming languages [4]. In this scenario, we need to consider how to build end-to-end workflows where, for example, simulations can be MPI applications written in Fortran or C/C++, and the analytics codes can be written in Java or Python (maybe parallelized by a Big Data framework).

In this work we introduce a new computing framework called IgnisHPC[1] to deal with that issue. The main goal of IgnisHPC is to unify in the same framework the development, combination and execution of HPC and Big Data applications using different languages and programming models. With this objective in mind, we can summarize the main contributions of IgnisHPC as follows:

- Unlike other frameworks such as Hadoop and Spark, IgnisHPC supports natively both JVM and non-JVM-based languages. Applications can be implemented using one or several programming languages following an API inspired by Spark's one.
- IgnisHPC uses MPI as backbone technology, which allows the framework to support many communication models and network architectures. In addition, MPI applications and libraries can be directly executed in an efficient way in IgnisHPC. In this way, most of the HPC scientific applications, which in many cases contain tens of thousands of lines of code, do not have to be ported to a new API or programming model. Nowadays, to the best of our knowledge, there is no other Big Data framework with that feature.
- In IgnisHPC, MPI codes can be easily combined with typical MapReduce operations to create hybrid applica-

*This work has been supported by MICINN (RTI2018-093336-B-C21, PLEC2021-007662), Xunta de Galicia (ED431G/08, ED431G-2019/04 and ED431C-2018/19) and the European Regional Development Fund (ERDF).

*Corresponding author

Email addresses: cesaralfredo.pineiro@usc.es (César Piñeiro), juancarlos.pichel@usc.es (Juan C. Pichel)

[1]It is publicly available at https://github.com/ignishpc

tions. Therefore, users can use the programming models that best fit their data-intensive and compute-intensive tasks in the same application.
– A thorough experimental evaluation has been carried out to demonstrate the benefits of IgnisHPC in terms of performance and productivity. The study showed that IgnisHPC clearly outperforms Spark when considering different types of Big Data application patterns. Moreover, we proved that running MPI (and hybrid MPI+OpenMP) applications from IgnisHPC is easy and as efficient as executing them natively.
– To avoid dependencies, IgnisHPC is fully containerized and supports some of the most well-known resource and scheduler managers.

The remainder of this paper is organized as follows. Section 2 provides some context about Big Data and HPC technologies. Section 3 explains in detail the architecture and modules of IgnisHPC. Section 4 describes how to implement applications using the IgnisHPC API. Section 5 focuses on the integration of MPI in IgnisHPC, and how MPI applications can be executed within the framework. The experimental evaluation is shown in Section 6. Section 7 discusses the related work. Finally, the main conclusions derived from this work are explained.

2. Background

2.1. Big Data frameworks

MapReduce is a programming model introduced by Google for processing and generating large data sets on a huge number of computing nodes [5]. Apache Hadoop [2] was the first open-source implementation of the MapReduce programming model. It was widely adopted by both industry and academia, thanks to that simple yet powerful programming model that hides the complexity of parallel task execution and fault-tolerance from the users. However, most applications do not fit this model and require a more general data orchestration.

Apache Spark [3] was designed to overcome some of the Hadoop limitations, especially when considering iterative jobs. Nowadays Spark is considered the *de-facto* standard for Big Data processing. Unlike Hadoop, Spark uses Resilient Distributed Datasets (RDDs) which implement in-memory data structures used to cache intermediate data across a set of nodes. Since RDDs can be kept in memory, algorithms can iterate over RDD data many times very efficiently. In addition, Spark provides many attractive features such as fault-tolerance, as RDDs can be regenerated through lineage when compute nodes are lost. Spark uses a thread-based worker model for executing the tasks. In this way, a Spark job is controlled by a driver program, which usually runs in a separate master node. On the other hand, the parallel regions in the driver program are shipped to the cluster to be executed.

Spark is implemented in Scala and also has interfaces to execute Java, Python and R applications. Both Hadoop and Spark are capable of running codes written in other programming languages, but they suffer performance issues since they require sharing data outside the Java Virtual Machine (JVM) through system pipes [6]. Contrary to the common belief, this behavior also applies to Python in Spark because, while the Python driver code can be executed within the JVM thanks to Jython [7], executors are directly executed with the available Python interpreter. As a consequence, the performance of Python codes is affected like any non-JVM language such as C++ .

In our previous work [8], we introduced Ignis, a first attempt to build an efficient and scalable multi-language Big Data framework. Despite being a prototype, Ignis has two significant contributions with respect to Spark. First, it allows to execute natively applications implemented using non-JVM languages such as Python and C/C++. Second, it supports multi-language applications, so different computing tasks could be implemented in the programming language that best suits them. However, there are also some important limitations as the following. For instance, Ignis is restricted to use TCP sockets for inter-node communication, so it is not possible to take advantage of typical HPC networks such as Infiniband, Aries [9], Slingshot [10], TofuD [11], etc. Another issue is that Ignis only supports one data partition per worker (executor). As a consequence, in order to work with big datasets, it is necessary to create new workers, which causes a degradation in the I/O performance. Ignis is completely containerized, but it uses a very limited ad-hoc solution for containers orchestration (named *Ancoris*). And finally, there was no submitter in Ignis, so jobs are manually launched using several configuration scripts.

2.2. MPI

Message Passing Interface (MPI) is the most widely used and dominant programming model in HPC. In MPI, processes make explicit calls to library routines defined by the MPI standard to communicate data between two or more processes. These routines include both point-to-point (two party) and collective (many party) communication. Note that from MPI-3 new features were introduced to enable MPI processes within an SMP node to collectively allocate shared memory for direct load and store operations, which enables the shared-memory-processes to more efficiently share data.

Among the different MPI implementations, the most successful ones are MPICH [12] and Open-MPI [13]. In particular, IgnisHPC uses MPICH as backbone technology to perform all the communications between processes, which allows the framework to support many communication models and network architectures (e.g. Infiniband, Slingshot, etc.). As we will explain later, one of the most important consequences of this design decision is that IgnisHPC can efficiently execute native MPI applications. Therefore, it is not necessary to port MPI applications using a different API. In this way, IgnisHPC is able to bring together the benefits of HPC and Big Data worlds into the same framework.

We have considered MPICH instead of other MPI implementations because it is a mature project and provides several important features that are critical in a Big Data environment. For example, it is possible to join processes dynamically. It means that MPICH allows to connect independent instances of an MPI process at runtime, which is essential for increasing the

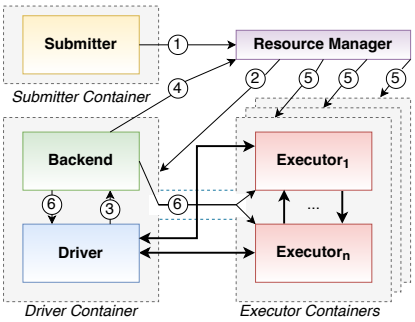


Figure 1: Scheme of the architecture of IgnisHPC.

number of processes when more parallelism is needed or for replacing lost processes when a computing node fails.

2.3. Resource managers and schedulers

In a Big Data environment, it is necessary to manage and balance the cluster resources to allow multiple applications and frameworks to be efficiently executed together on the same system. That is the goal of resource managers such as Apache Mesos [14] and Nomad [15].

In particular, Apache Mesos groups all the physical resources of each node of the cluster and make them available for the applications as a single pool of resources. Among the features of Mesos we can highlight that it provides resource isolation thanks to its support to Docker containers [16]. So, it allows the execution of jobs in a custom independent environment both in terms of resources and installed software.

Users cannot interact directly with Mesos because it is only a resource planner, so an orchestration framework is required to run and schedule tasks. We can find many orchestration frameworks depending on the type of tasks to be executed. In this work we have considered two of the most relevant, Apache Marathon [17] and Apache Singularity [18], which both support Docker containers orchestration.

Finally, Nomad is a simple workload orchestrator created by HashiCorp. Its flexible and consolidated workflow provides users the functionality of both a resource and a scheduler manager, combined into a single system. Nomad is a modern alternative to Mesos and also supports containerized and non-containerized applications with different life cycle. Unlike old legacy platforms, it is prepared for running heterogeneous applications and using accelerators such as GPUs in a simple way.

3. IgnisHPC

3.1. Architecture of the framework

We can divide the IgnisHPC architecture into four independent modules which run inside Docker containers: Submitter, Backend, Driver and Executor. A scheme is shown in Figure 1. Modules were implemented in different languages, using Apache Thrift[2] for the inter-module communications. In the figure, thin arrows represent those RPC communications, bold arrows data transfers, and numbers indicate the call order.

The Submitter module is in charge of launching the IgnisHPC jobs (1) making a request to the resource manager (and scheduler), which is an external dependency that is responsible of the containers orchestration and the management of their resources. Afterwards, the driver container is started (2), where the Driver module is the container entrypoint. That module exposes all the available features of IgnisHPC through a user API created as interface to the Backend, where the API logic is implemented as a service (3). The Backend module is started inside the driver container after the driver code initializes the framework. Since Driver and Backend are in the same container, they share the same resources. The Backend is responsible of making the requests to the resource manager following the instructions specified in the driver code (4). As a consequence, the resource manager will create the executor containers (5). The Executor module contains the low-level implementation of a set of operations required by the Backend for each supported programming language. Note that IgnisHPC uses an SSH tunnel to connect driver and executor containers to encrypt and protect the communications. Finally, the Backend is connected to the executors in order to perform the low-level API operations (6). It is important to highlight that the Driver is also considered an executor by the Backend to handle data transfers. Note that only large data transfers are performed using MPI, otherwise RPC is used.

There are important architectural differences between IgnisHPC and Ignis [8], our first prototype of a multi-language Big Data framework. Changes performed to IgnisHPC focused on removing some important limitations and performance issues shown by Ignis (see Section 2.1). Among them we can highlight the following:

- One of the main goals of IgnisHPC is to unify the execution of Big Data and HPC workloads in the same framework. For this reason data transfers in IgnisHPC (bold arrows in Figure 1) are performed using MPI. It has enormous advantages over the inter-node communications with TCP sockets used by Ignis. First, IgnisHPC supports many different communication models and network architectures (e.g. Infiniband, Slingshot, etc.). In this way, it covers the characteristics of the vast majority of Big Data and/or HPC clusters. Moreover, MPI applications and libraries can be directly executed in IgnisHPC. It means that HPC scientific applications, which in many cases contain tens of thousands of lines of code, do not have to be ported to the IgnisHPC API. And finally, it is possible to combine in the same multi-language code HPC tasks (using MPI) with Big Data tasks (using MapReduce operations).
- IgnisHPC has a new Submitter module that handles jobs using an external resource manager. This module includes a submit script, similar to Spark's `spark-submit`, that allows users to easily launch IgnisHPC jobs. On the contrary, jobs in Ignis are manually configured and launched using several scripts. There is no Submitter

[2] https://thrift.apache.org

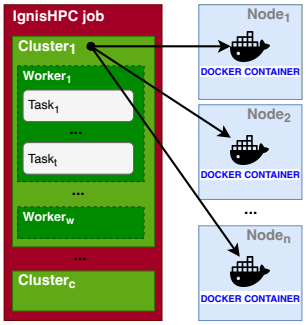


Figure 2: Job hierarchy in the IgnisHPC framework.

module in its architecture.

- Ignis used a MANAGER module (one per executor container) that acted as middleman between the BACKEND module and the executors. To be more efficient, IgnisHPC removed that module and its functionalities were adopted by the BACKEND.
- IgnisHPC supports some of the most well-known resource and scheduler managers. In addition, it was designed to easily add new managers. However, Ignis is bonded to work only with an ad-hoc manager, which has limited functionalities.

3.2. Jobs in the framework

It is important to know the structure of an IgnisHPC job. It consists of a set of Docker containers distributed in multiple computing nodes grouped in *Clusters* (see Figure 2). *Workers* are bonded to a single programming language and run inside a *Cluster*, so at least one *Worker* has to be created for each programming language in order to build multi-language applications. A *Worker* instantiates at least one process (executor) on each Docker container with the aim of executing its tasks in parallel, processing them as a pipeline.

Clusters are independent, each one has its own assigned resources, so they can execute different tasks at the same time. Using multiple *Clusters* could be useful if stages or phases of the job have some compatibility issues. In this way, incompatible phases would be executed by differently configured *Clusters*. However, *Workers* can be executed in shared mode (disabled by default). It means that executors of two or more *Workers*, which are located in the same container, would share the available resources. Normally users configure each *Worker* to use a part of the *Cluster* resources (e.g. cores, memory, etc.).

3.3. Resource manager

Since IgnisHPC must be executed inside Docker containers, a resource and scheduler manager is required to handle the cluster resources and launch these containers. Note that any framework that meets these requirements could be used in IgnisHPC by implementing a basic interface. Currently IgnisHPC supports the following managers:

- *Docker*: It is the easiest way to run IgnisHPC locally on a single machine. It uses the Docker client to directly launch containers.
- *Ancoris* [8]: This is the only one supported by Ignis and it was designed as a simple and light resource manager. It executes itself inside Docker containers and is composed of two types of instances: master and slaves. The master manages the available resources in the cluster, and it is responsible of launching the containers. Slaves expose the resources of their host to the master when they are deployed.
- *Mesos+Marathon*: Since Spark supports Mesos, using it as resource manager together with Marathon as container orchestrator allows users to execute in a single environment Spark and IgnisHPC jobs. In addition, since IgnisHPC is able to execute efficiently Big Data and MPI-based applications, we are merging both Big Data and HPC software ecosystems in just one execution environment.
- *Mesos+Singularity*: The same benefits commented above apply to the combination of Mesos and Singularity.
- *Nomad*: It combines in the same framework a resource and a scheduler manager. Due to its lack of dependencies, it is the best option to install in a cluster from scratch. Moreover, it has better support for devices like GPUs than Mesos, which allows to create heterogeneous execution environments.

It is important to highlight that the cost of deploying Docker containers is very low. For instance, considering Nomad, it is possible to deploy thousands of Docker containers in just a few seconds[3].

3.4. Driver module

The DRIVER module is a user API that allows access to all IgnisHPC functionalities. The driver program describes the high-level control flow of the application, and it can be programmed in any of the supported languages (currently, Java, Python and C/C++). The DRIVER was designed as a Thrift RPC interface to the BACKEND so the logic has not to be re-implemented for every programming language. More details about the driver API and how to implement an application in IgnisHPC are provided in Section 4.

3.5. Backend module

The BACKEND module contains the services that define the DRIVER's logic. For instance, the `reduceByKey` function requires searching and grouping the keys. These operations are defined in the BACKEND, but they are implemented in the EXECUTOR module for a specific programming language.

The BACKEND module is also responsible of sending requests to the resource manager in accordance with the instructions specified in the driver code. These instructions are lazily executed, so the BACKEND registers the function calls as a task dependency graph. When a task that represents an action in the driver code is created (e.g., a call to `count`), all the tasks in its dependency graph are executed. An example is shown in Figure

[3] `https://www.hashicorp.com/c1m`

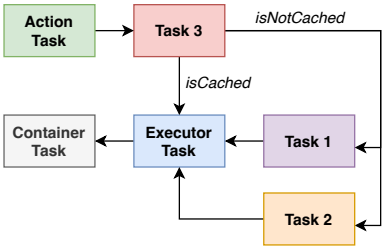


Figure 3: Example of a task dependency graph.

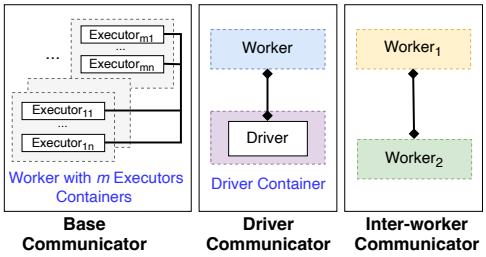


Figure 4: MPI communicators in IgnisHPC.

3, where the *Action Task* depends on *Task 3*, and *Task 3* depends on *Task 1* and *2*. Note that a task dependency is only computed if the task was never executed or if its result was not explicitly cached. The *Executor* and *Container* tasks are always executed as final dependencies. These tasks check that executors and containers are running and ready to be used.

Finally, IgnisHPC is able to recover after a failure of a cluster node or some of the executors. Affected tasks are traced by the Backend in such a way that only their executors are reallocated and recomputed. If the affected tasks are cached, the recovery process will be faster since it is not necessary to recalculate their dependencies. Note that this process is automatic, but users can tune the recovery process using the persistence functions in the driver code (see Section 4).

3.6. Executor module

The Executor module implements the operations defined by the Backend, where each supported programming language has its own implementation. In order to add support for a new language in IgnisHPC, a minimum implementation only requires programming the context class. The executor context allows the API functions to interact with the rest of the IgnisHPC system. In this way, among the functionalities of the context we find the exchange of user variables between driver and executors or providing the executors access to the MPI communicators.

As we explained previously, IgnisHPC uses MPI (that is, MPI communicators) to perform all the communications related to the Executor module. IgnisHPC constructs three types of communicators for data transfers (see Figure 4):

- *Base communicator*: for each worker there is a communicator that includes all its executors. This communicator always exists. If one executor is lost, the communicator is destroyed and a new communicator is created including a new executor. To that end, the capability of linking dynamically a single process to a communicator introduced in MPI-3 was of special importance. Without this feature all processes would have to be launched at the same time, and in case a process died, it could not be replaced causing the job to fail.
- *Driver communicator*: it joins a base communicator to the driver process. It is created when the driver and a worker exchange data.
- *Inter-worker communicator*: it is created joining the base communicators of two workers. It is used to send data from one worker to another and it will be destroyed as soon as one of the two workers stops its execution. This communicator is only created between workers that execute the operation `ImportData`.

The base communicator for each worker is accessible to programmers by the executor context. It means that IgnisHPC functions can be implemented using that communicator and MPI primitives (e.g. gather, scatter, broadcast, reduce, etc.). As a result, IgnisHPC supports the execution of pure MPI applications with minimal modifications in the original code. A detailed explanation is provided in Section 5.

Another benefit of using MPI for data transfers is the performance improvement of iterative applications. When using Big Data frameworks such as Ignis and Spark, an iterative application requires the driver to perform an evaluation task per iteration to obtain the final result. Each evaluation has three steps: stopping the executors, analysis of the partial results by the driver and restarting the executors. Note that starting and stopping the executors is very costly in terms of performance. IgnisHPC avoids the driver evaluations because executors share the partial results of each iteration using their MPI base communicator. Therefore, it is not necessary to stop them because they do not need to wait for the driver. This has even a bigger impact on performance for those applications with many short iterations.

3.7. Submitter module

The Submitter is an IgnisHPC module consisting of a set of scripts and utilities for configuring and launching jobs. As we commented previously, Ignis had no module to launch tasks, and the Driver module was launched manually using *Ancoris* (see Section 3.3). The Submitter is a container, which can be accessed by SSH. There users can set up jobs in a similar environment where the IgnisHPC applications will be executed.

The main utility of the Submitter module is the `ignis-submit` script that, like `spark-submit`, allows users to launch IgnisHPC jobs in the cluster. The script only requires as mandatory arguments a Docker image and the driver program. There are also the following optional parameters:

- `name`: a job name can be specified.
- `arguments`: after the driver program name, all the parameters will be considered as driver arguments.
- `attach mode`: by default, jobs are launched in unattached mode. That is, `ignis-submit` launches the job and exits. Attach mode allows users to control the job as if the

```
# Python driver
ignis-submit ignishpc/python python3 mydriver.py

# C++ driver
ignis-submit --name myapp --properties
     ignis.driver.memory=2GB ignishpc/cpp ./mydriver 0 -g 2
```

Figure 5: Job submission examples.

driver runs locally, so output is printed in real time, and it is possible to manually kill the job.
– `properties`: users can change the default properties before launching the DRIVER module. EXECUTOR properties can be redefined later but DRIVER properties are set only by `ignis-submit`.

Figure 5 shows two job submission examples. The first one is a Python basic submission with only its base image and the driver application. The second one deals with a C++ driver and optional parameters. In particular, `--name` sets the job name, `--properties` changes the driver default memory to 2 GB, and `0 -g 2` are considered arguments of `mydriver`. Note that `ignishpc/cpp` is the C++ base image.

3.8. Data storage

IgnisHPC provides multiple options for data storage. Users can choose a type of storage according to their particular execution environment. Storage must be defined as a property before the worker creation. IgnisHPC supports the following storage options:

– *In-Memory:* it is the best performer since all data is stored in memory. Memory consumption could be an issue, so it is not suitable for all kinds of jobs.
– *Raw memory:* data is stored in a memory buffer using a serialized binary format. Extra memory consumption is minimal and the buffer is compressed by Zlib [19], which has nine compression levels. Level six is applied by default, but it can be changed when the worker properties are defined.
– *Disk:* similar to raw memory but the buffer is stored as a POSIX file. Performance is much lower but it allows to work with large amounts of data that cannot be completely stored in memory.

IgnisHPC behaves very similar to Spark in terms of data locality. Data is split into several partitions which are assigned to executors. Each partition is assigned to a single executor. In case another executor needs that partition, it will be sent using MPI. By default, IgnisHPC stores all partitions in memory to achieve the best possible performance. If there is not available memory, Docker sends some data to the container swap automatically. The user can modify this policy by changing the swap size or removing it. In case the data size exceeds the swap or the swap performance is insufficient, IgnisHPC allows users to set disk as primary storage, so partitions will always be stored on disk. In this case partitions will only be loaded into memory at processing time.

On the other hand, there is an important difference in how memory is handled by Ignis and IgnisHPC. Since Ignis was just a prototype, for simplicity in the implementation, it assigns one data partition to each executor. In this way, if it is necessary to increase the partition size, a realloc operation is performed in such a way that the complete partition is copied to a different memory location. The consequence is a noticeable increment in the memory consumption. This restricts Ignis to work with smaller input datasets. IgnisHPC overcomes that limitation supporting several data partitions per executor. Note that an executor can spawn several threads to process the data partitions in parallel.

Type	Functions
Conversion	`map, filter, flatmap, keyBy, mapPartitions, keys, values, mapValues`, etc.
Group	`groupBy, groupByKey`
Sort	`sort, sortBy, sortByKey`
Reduce	`reduce, treeReduce, aggregate, treeAggregate, fold, reduceByKey, aggregateByKey`, etc.
I/O	`collect, top, take, saveAsObjectFile, saveAsTextFile, saveAsJsonFile`, etc
SQL	`union, join, distinct`
Math	`sample, sampleByKey, takeSample, count, max, min, countByKey, countByValue`
Balancing	`repartition, partitionBy`
Persistence	`persist, cache, unpersist, uncache`

Table 1: Example of some IDataFrame functions supported by IgnisHPC.

4. Programming applications for IgnisHPC

IgnisHPC requires a minimal driver code that implements the application at high-level. To facilitate the adoption from the Big Data community, the IgnisHPC API is inspired by the Spark API in such a way that IgnisHPC codes are easily understandable by users who are familiar with Spark. In comparison to Ignis, we have extended the API to cover the most important primitives required by Big Data applications. For instance, IgnisHPC includes functions such as `join` and `union` for graph processing. The IgnisHPC driver API is composed by six main classes:

– `Ignis` starts and stops the driver environment.
– `IProperties` defines the execution environment properties.
– `ICluster` represents a group of executors containers. It is possible, for example, to execute remote commands (`execute`, `executeScript`) and send files (`sendFile`, `sendCompressedFile`) to them.
– `IWorker` represents a group of processes of the same programming language. This class includes functions to read files (`textFile`, `partitionJsonFile`, etc.), import data partitions from another worker (`importData`), send data from the driver (`parallelize`) and execute external codes (`loadLibrary`, `call`, `voidCall`). As we explain later, the former routines allow IgnisHPC to execute MPI applications within the framework.

```python
#!/usr/bin/python

import ignis

# Initialization of the framework
ignis.Ignis.start()
# Resources/Configuration of the cluster
prop = ignis.IProperties()
prop["ignis.executor.image"] = "ignishpc/full"
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
worker_python = ignis.IWorker(cluster, "python")
# Task 1: Tokenize text into pairs (x, y)
# The edges are stored in reversed order
tc = worker_python.textFile("graph.dat")
edges = tc.map(lambda x_y: x_y.split(" ")[::-1])
# Initialization of a C++ Worker
worker_cpp = ignis.IWorker(cluster, "cpp")
# Transfer data from Task 2 - Python
tc2 = worker_cpp.importData(tc).cache()

oldCount = 0
nextCount = tc.count()
while True:
  oldCount = nextCount
  # Task 2: Perform the join,(y, (z, x)) pairs,
  # to obtain the new (x, z) paths.
  new_edges = tc2.join(edges).
     map("libexample.so:Reverse2")
  tc2 = tc2.union(new_edges).distinct().cache()
  nextCount = tc2.count()
  if nextCount == oldCount:
    break
# Show result
print("TC has %i edges" % tc2.count())
# Stop the framework
ignis.Ignis.stop()
```

Figure 6: *Transitive Closure* driver code in Python.

- `IDataFrame` contains all the functions of the MapReduce paradigm, similarly to Spark RDD. A function can be a transformation that generates another IDataFrame or an action that generates a final result (see Table 1). With respect to Ignis, besides the support for new API functions, IgnisHPC has increased the overall performance for some types of routines (e.g., *Group* and *Sort*) thanks to its complete redesign using MPI.
- `ISource` is an auxiliary class used by meta-functions such as `map` in the driver. This class acts as a wrapper for the input parameters. It is also used to store variables and send them to the executors. Those variables can be obtained by the executors using the context.

Note that all the API operations that move data between executors perform an internal shuffling operation (e.g., `parallelize`, `collect`, `partitionBy`, etc.).

4.1. An example: Transitive Closure

With the goal of illustrating how applications are programmed in IgnisHPC, Figure 6 shows an example of a driver implemented in Python for computing the *Transitive Closure* of a graph. This algorithm finds out if a vertex x is reachable from another vertex y for all vertex pairs (x, y) in the graph. Note that an equivalent driver code could be implemented in any of the IgnisHPC supported languages (C/C++ and Java) using a similar syntax.

```cpp
#include<ignis/executor/api/function/IFunction.h>

using namespace ignis::executor::api;
typedef std::pair<int64_t, int64_t> ipair;
typedef std::pair<int64_t, ipair> ipair2;

// Reverse pairs (z, (x, y)) to (z, (y, x))
class Reverse2 : public function::IFunction<
      ipair2, ipair2>{
  ipair2 call(ipair2& x_y, IContext& context){
      return {x_y.second, x_y.first};
};

ignis_export(Reverse2, Reverse2)
```

Figure 7: Function in C++ used by a `map` operation for the *Transitive Closure* application.

First, the IgnisHPC framework is initialized (line 6). Properties are created to configure and build a cluster (lines 8 to 14). Note that the properties definition is optional, and IgnisHPC could read them from a default configuration file. Moreover, IgnisHPC introduce the possibility of overwrite the default values when a job is submitted like Spark using the new Submitter module. Therefore, the Docker image, the number of containers, the number of cores and the memory per container could be defined out of the driver code. Computing the Transitive Closure has two phases. To illustrate the multi-language support in IgnisHPC, each phase was implemented in a different programming language. The first one uses a Python executor and the second a C++ executor. The first stage consists of a `map` operation that takes as input a text file and creates pair values that represent edges in the graph (line 20). As a consequence, it is necessary to previously create a Python worker in the cluster (line 16). It is important to note that creating the worker is mandatory and is not related to the driver programming language. On the other hand, if the worker and the driver code are in the same language, lambda functions can be used (line 20).

The following phase of the algorithm is iterative: edges are joined into a path until there are no new paths. Since this phase is implemented in C++, a C++ worker should be created (line 22). Data is shared between workers using the `importData` function (line 24). The driver code ends printing the results. The framework must be stopped before the driver ends (line 41) to stop the backend. Unlike Ignis, IgnisHPC automatically detects when the driver process ends and stops it. However, this is not a good practice.

In IgnisHPC a lazy evaluation is performed when a result is not required explicitly. In the example, the trigger that causes the tasks to be launched are the calls to the `count` function. This approach is also followed by Spark where RDDs are computed lazily the first time they are used in an action [20].

Most of the driver functions are meta-functions. That is, generic functions that require another one to perform an internal operation. This is the case of `map` in the example of Figure 6 (lines 20 and 32-33). To implement those functions we should use the executor API provided by IgnisHPC. Basically, it defines a simple interface based on the number of required input parameters. Figure 7 shows an example corresponding to the C++ function used by `map` in the driver code. Since `map` takes one parameter and also returns one parameter, `IFunction` is

```
1  // Python lambda
2  data.reduce("lambda a, b: a + b")
3
4  // C++ lambda
5  v = 2
6  data.reduce(
7   ISource("[](int a, int b, IContext &c){      \
8            return (a + b) * c.var<int>(\"v\"); \
9        }").addParam("v",v)
10 )
```

Figure 8: Examples of text lambda for a Python and a C++ executor.

used. In case there are two input parameters (e.g., `reduce`), `IFunction2` would be used, and so on. Note that if the function does not return any value (e.g. `foreach`), functions have the same name but with the `Void` prefix.

4.2. Text lambda functions

As explained previously, lambda functions need that driver and executor codes were implemented in the same language because native code serialization is required. We refer to code serialization as the process by which a function or set of instructions are converted into bytes to be sent and executed in a different environment. Note that although Python and Java are able to serialize code both languages face compatibility problems. On top of that, C++ does not allow any type of code serialization. To overcome these limitations IgnisHPC implements its own multi-language lambdas without source code serialization, named *text lambdas*. In this way, IgnisHPC allows to define lambdas as text, using the executor language syntax. The executor will transform the lambda text into source code to be used as a meta-function parameter. It is important to highlight that the language of the driver code is indifferent.

Figure 8 shows an example of a text lambda that accumulates all elements (line 2) used by a `reduce` function. It uses Python syntax so must be evaluated by a Python executor. Another example is shown in line 7. It defines a text lambda that captures the value of a variable, which is read from the context. This lambda function will be compiled and loaded by a C ++ executor. Performance is not affected by using text lambda functions but it can add some overhead to the compilation process, especially for C++.

In the same way, using the mechanism that allows to execute text lambda functions, IgnisHPC can send a complete job or application to the executors. This is possible thanks to `loadLibrary`, which can be used to execute a full source code as an IgnisHPC library. It has again a small impact on the compilation time. More details about `loadLibrary` are provided in Section 5.2 using MPI applications as use case.

5. MPI on IgnisHPC

Our first prototype, Ignis, was limited to perform inter-process communications using only TCP sockets (different computing nodes) or shared memory (same node). However, IgnisHPC was completely redesigned to use MPI as backbone technology. As a consequence, all communications are internally implemented by MPI routines. As we explained previously in the paper, this change makes it possible for IgnisHPC to support more communication models and network architectures. In addition, an important advantage of our approach is that, once IgnisHPC has configured the MPI communications, users can combine in the same application pure MPI libraries using the IgnisHPC communicators together with standard Big Data functions (`map`, `reduce`, `collect`, etc.).

5.1. Integration of MPI into a Big Data environment

MPI was not designed to run on Docker containers. As a consequence, there are several problems that should be addressed. First, by default and to preserve an isolation runtime environment, Docker creates a private virtual network between the host and the containers. If two containers are launched on the same host, we can execute MPI processes in the same way that they were two real nodes of a cluster. But if we launched those containers on different hosts, the communication is impossible since they belong to different networks. We can find in the literature several works that deal with this issue (see the Related Work section). For instance, some approaches opt for launching the container on the host network or creating a virtual network between the cluster nodes [21]. However, these configurations are difficult to handle and implement by resource managers in Big Data environments.

Second, there are important differences in how ports are handled by a Big Data or an HPC environment. For instance, ports are considered a resource in a Big Data environment because there are services that require an exclusive port, which is not the case in HPC. MPI needs ports to establish connections between processes but restricted to a range. However, ports provided by resource managers in a Big Data environment are usually random and not consecutive.

Finally, IgnisHPC can internally spawn several threads per MPI process (executor) to increase the performance when processing and/or communicating data. All these threads use communicators to exchange data in parallel. Every time a communicator is created, MPI assigns a virtual interface to it. However, a virtual interface can only be used by one communicator at the same time, so parallel communications require the use of multiple virtual interfaces. In the most recent MPICH version, which is the MPI implementation used by IgnisHPC, virtual interfaces are assigned sequentially. Since IgnisHPC creates and destroys communicators dynamically, it is not possible to assure that threads can always exchange data in parallel using communicators with different assigned virtual interfaces. The consequence is a degradation in the performance.

To overcome the above problems, IgnisHPC applies the following changes to MPICH:

- Containers: MPICH has been designed to work on a local network. Docker containers can be joined to a network but only within the same node (internal network). Although resource managers can export a service from the internal network to the local network, this causes a problem when MPI is executed containerized. MPICH uses a service to store the network addresses of the launched

```
// mpi.h
#ifndef IGNIS_MPI
#define IGNIS_MPI

#include_next <mpi.h>

extern MPI_Comm IGNIS_COMM_WORLD;
#undef MPI_COMM_WORLD
#define MPI_COMM_WORLD IGNIS_COMM_WORLD

#endif
```

Figure 9: C header that replaces `MPI_COMM_WORLD` by `IGNIS_COMM_WORLD` (global variable).

MPI processes, but when using containers, each MPI process stores the values corresponding to the internal network which are not valid outside the node. For this reason, it is necessary to modify MPICH in order to store the correct network values that correspond to the local network.

- Ports: now MPICH uses a list of ports provided by the resource manager instead of a range.
- Multithreading: MPICH was modified to assure that all threads use a different virtual connection. In this way, communications between threads can always be performed in parallel.

5.2. *Running MPI applications in IgnisHPC*

IgnisHPC can execute MPI applications implemented in any of the supported languages. We must highlight that, to the best of our knowledge, currently there does not exist another Big Data framework with this feature. Most of the MPI codes for HPC are implemented in C/C++, while applications in other languages such as Java and Python are a minority. So although IgnisHPC supports Python and Java, we will focus on C/C++ MPI applications.

To explain how to execute an MPI application within IgnisHPC, we have considered LULESH [22] as guiding example. LULESH is a proxy HPC application for shock hydrodynamics with more than 5,000 lines of C++ code.

MPI applications, like other IgnisHPC codes, must be implemented using the executor API to be used from the driver (see Figure 7). However, some minimal modifications should be previously applied to the original MPI codes:

- MPI initialization: IgnisHPC controls the MPI environment, so `MPI_Init` and `MPI_Finalize` must be removed from the MPI application.
- `MPI_COMM_WORLD`: MPI applications use a default communicator but IgnisHPC requires its own communicator. The simplest solution is to create a custom header to overwrite the default communicator. Figure 9 shows an implementation of this functionality.
- I/O data: These modifications are optional. In some scenarios, it is interesting to allow IgnisHPC to handle the operations on input and output data of an MPI application. This is the case, for example, when the output file of the application will be afterwards processed by other IgnisHPC tasks. If IgnisHPC manages the output file, data is kept in memory. If not, the output file would be written to disk and read again to continue executing the following tasks, causing an important degradation in the performance. To do that, read and write functions related to input/output files will be removed from the MPI source code. As we explain next, they will be replaced by input and return parameters of the `call` function in the corresponding executor code.

```
1  using ignis::executor::api;
2
3  extern int main(int argc, char *argv[]);
4  MPI_Comm IGNIS_COMM_WORLD;
5  class Lulesh : public function::IVoidFunction0{
6  public:
7        void call(IContext &context) override {
8
9        IGNIS_COMM_WORLD = context.mpiGroup();
10       const char *argv[100];
11       argv[0] = "ignis-cpp";
12       int argc = 1;
13
14       // Start parsing arguments ***************
15       if (context.isVar("i")) {
16         argv[argc++] = "-i";
17         argv[argc++] = const_cast<char *>(
18             context.var<std::string>("i").data());
19       }
20
21       if (context.isVar("s")) {
22         argv[argc++] = "-s";
23         argv[argc++] = const_cast<char *>(
24                context.var<std::string>("s").data());
25       }
26
27       ...
28
29       if (context.isVar("h") && context.var<bool>("h")) {
30         argv[argc++] = "-h";
31       }
32       // End parsing arguments ***************
33
34       main(argc, const_cast<char **>(argv));
35       }
36 };
37
38 ignis_export(lulesh_app, Lulesh)
39
40 create_ignis_library("lulesh_app");
```

Figure 10: Executor code for LULESH using C++.

Figure 10 shows the executor code for calling LULESH from the IgnisHPC driver. First, the global variable for the communicator of Figure 9 is created (line 4) and initialized with the IgnisHPC MPI group (line 9). LULESH is a benchmark, so it does not receive any data from IgnisHPC. As a consequence, according to the executor API explained in Section 4, it is of type `IVoidFunction0` (line 5). The only mandatory method to be implemented is `call` (line 7). Inside that method, the application arguments are parsed from the IgnisHPC context. In our example, each argument is individually parsed to create a user friendly interface. However, the arguments could be parsed together as a list, reducing noticeably the necessary lines of code. The `call` method ends calling the LULESH main function (line 34). Afterwards, `Lulesh` class is exported (line 38). This operation is only necessary in C++. To finalize, an IgnisHPC library is created (line 40), which will be used to call LULESH from the driver.

Finally, we show how to use an MPI application from the

```
worker.loadLibrary("liblulesh.so")
worker.voidCall("lulesh_app", s = "70", i = "2400")
```

```
worker.loadLibrary("liblulesh.so");
worker.voidCall(ISource("lulesh_app").
                addParam("s","70").
                addParam("i","2400"));
```

Figure 11: Lulesh usage from a Python driver (top) and its equivalent C++ code (bottom).

```
...
# Initialization of a Python Worker
worker = ignis.IWorker(cluster, "python")
# Load wordcount function implemented in Python with MPI
worker.loadLibrary("wordcount.py")
# Task 1: Read a text line by line
text = worker.textFile("text.txt")
# Task 2: Split each line into words
words = text.flatmap(lambda line : line.split(" "))
# Task 3: Execute wordcount using words as input
counts = worker.call("wordcount", words)
# Task 4: Result is stored as json
counts.saveAsJson("result.json")
....
```

Figure 12: Wordcount example as MPI hybrid application.

driver. The example of Figure 11 focuses only in the necessary functions to execute LULESH using a Python and a C++ driver. Note that the MPI application should be previously compiled as a library (`liblulesh.so`). The example assumes that there is a C++ worker in the driver where two functions are executed: `loadLibrary` and `voidCall`. On the one hand, `loadLibrary` loads all the classes from the library declared in `create_ignis_library`. In this case, `Lulesh` is the only existent class. On the other hand, `voidCall` is an action that causes the execution of the library. If the library returns an output to IgnisHPC, `voidCall` should be replaced by `call` that would return an `IDataFrame` object. Library arguments in C++, which were parsed in the executor code, are added to the function using `addParam` from the `ISource` class. This syntax could be also used in Python. Nevertheless, keyword arguments in Python are a more elegant alternative (see line 2 in Figure 11).

5.3. Hybrid applications

In IgnisHPC an MPI code can be combined with typical MapReduce operations to create a hybrid application. In this way, the different computing tasks could be implemented in the programming model and language that best suits them.

Figure 12 shows a simple example of a Wordcount application where an MPI Python library is combined with IgnisHPC API functions. Input data is distributed and prepared by IgnisHPC (Tasks 1 and 2), so MPI is only responsible of the compute-intensive part (Task 3). Observe that for using functions included in a Python library it is only necessary to load the library (line 5) and invoke the `call` routine with the name of the desired function (line 11). Finally, results are converted and written to disk in *json* format using the IgnisHPC API (Task 4). Figure 13 shows another example. In this case, API functions are combined with explicit calls to MPI routines (line 9) with the aim of creating the hybrid application.

```
# Each executor iterates its partitions to compute
# the partial sum. An MPI reduction between executors
# is performed to calculate the final result
def sumNumbers(parts, context):
 value = 0
 for part in parts:
  for n in part:
   value += int(n)

 value = context.mpiGroup().reduce(value, MPI.SUM)
 if context.executorId() == 0:
  return [[value]]
 return []

...
# Initialization of a Python Worker
worker = ignis.IWorker(cluster, "python")
# Task 1: Read a text line by line
numbers = worker.textFile("numbers.txt")
# Task 2: Find the sum of an array of numbers
result = numbers.mapExecutor(sumNumbers)
...
```

Figure 13: Sum of an array example as MPI hybrid application.

Operators	Batch (one pass)		Iterative (caching)		
	MB	TS	KM	PR	TC
textFile	✓	✓	✓	✓	✓
map	✓	✓	✓	✓	✓
mapValues	–	–	–	✓	–
flatMap	–	–	–	✓	–
reduceByKey	–	–	✓	✓	–
collectAsMap	–	–	✓	–	–
repartition	–	✓	–	✓	–
count	–	–	–	✓	✓
join	–	–	–	✓	✓
union	–	–	–	–	✓
distinct	–	–	–	–	✓
importData (I)	✓	–	–	–	–
sort	–	✓	–	–	–
saveAsTextFile	✓	✓	✓	–	–

Table 2: Operations used in each Big Data application. Minebench (MB), Terasort (TS), K-means (KM), PageRank (PR) and Transitive Closure (TC). Operators annotated with I are specific only to IgnisHPC.

6. Experimental evaluation

6.1. Experimental setup

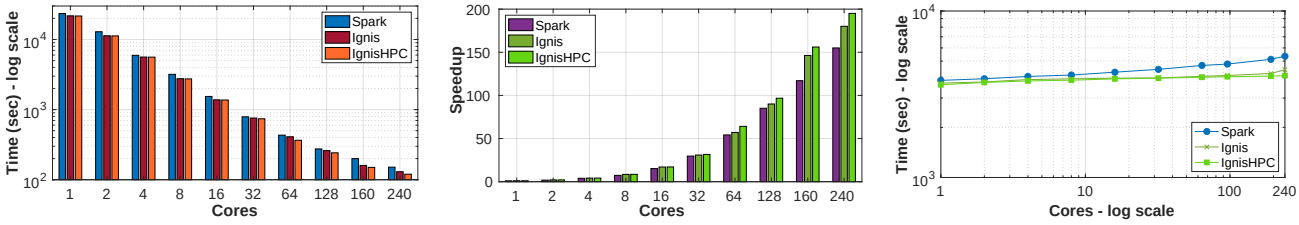
The experiments shown in this section were carried out on a 12-node cluster, where each node consists of:

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

It is a Linux cluster running CentOS 7 (kernel 3.10.0), Docker 20.10.2-ce and Spark 2.2.0 (with YARN [23] as cluster manager). Ignis and IgnisHPC run on an Ubuntu 20.04 image with MPICH 3.4.1.

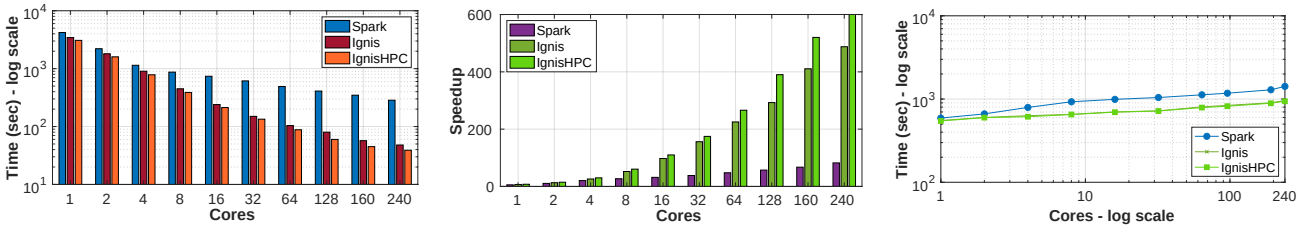
6.2. Big Data applications

We have selected five workloads that represent different types of application patterns for which Spark is considered the best performing Big Data framework [24]: *Minebench*, *TeraSort*, *K-Means*, *PageRank* and *Transitive Closure*. Table 2 lists the use of the most important operators by each Big Data application,



(a) Times (strong scaling) (b) Speedup (c) Times (weak scaling)

Figure 14: Study of the scalability of IgnisHPC, Ignis and Apache Spark running the Python Minebench application.



(a) Times (strong scaling) (b) Speedup (c) Times (weak scaling)

Figure 15: Study of the scalability of IgnisHPC, Ignis and Apache Spark running the Python & C++ Minebench application.

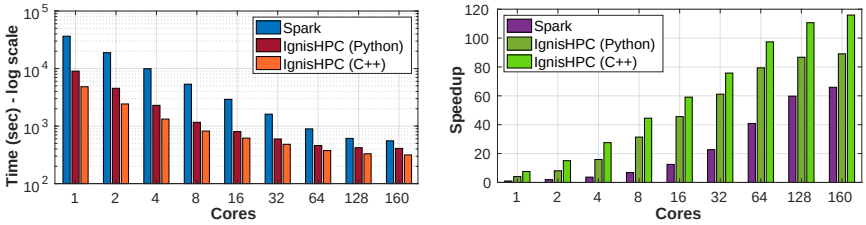
including basic core operators and specific ones implemented by the IgnisHPC framework.

– *Minebench (MB).* This application[4] performs the calculation of SHA-256 hashes imitating the Proof-of-Work algorithm used in the Bitcoin protocol [25]. Do not confuse it with the data mining benchmark suite NU-MineBench. 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 [26] is calculated for those transactions and its Merkle root hash is added to a block header. The second stage calculates the hash of the block header iteratively while the condition is not met. The strong scaling tests were obtained using an input file containing 300K blocks (120MB), while the weak scaling experiments fixed the input data per core to 300K blocks.
– *TeraSort (TS).* It is a sorting algorithm suitable for measuring the I/O and the communication performance of the considered frameworks. Elements in IgnisHPC are sorted by means of the MergeSort algorithm where elements are distributed by a regular sampling among the executors [27]. Note that this task requires that executors exchange data. The input data used in the tests contains 1 TB of text with 1.8B lines.
– *K-Means (KM).* This is a classic machine learning algorithm for data clustering, and it is a good example of an iterative MapReduce application. This pattern covers a large set of iterative machine learning algorithms such as linear regression, logistic regression, and support vector machines. The goal of KM is to classify a given data set through a certain number of clusters (K clusters). In each iteration, a data point is assigned to its nearest cluster center, using a map function. Data points are grouped to their center to further obtain a new cluster center at the end of each iteration (reduce). The experimental evaluation was carried out using the NUS-WIDE dataset [28], which contains 269,648 images with 500 attributes per image. In the tests the results were obtained after 10 iterations and $K = 81$.
– *PageRank (PR).* It is a graph algorithm which ranks elements by counting the number and quality of links. To evaluate the PR algorithm on IgnisHPC and Spark we used the LiveJournal graph from the SNAP repository [29], which contains 4.8M vertices and about 69M edges.
– *Transitive Closure (TC).* One of the most basic questions that arises when analyzing a complex graph G is whether one vertex x can reach another vertex y via a directed path. A way to store this information is to construct another graph, such that there is an edge (x, y) in the new graph if and only if there is a path from x to y in the input graph. This new graph is called the Transitive Closure of G. Since computing the TC is very costly, we used a small graph with 75 vertices and 200 edges in our tests.

6.2.1. Analysis and discussion

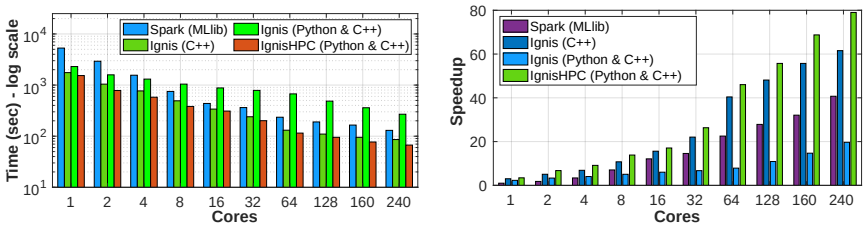
We now present the performance results from our evaluation of Spark, Ignis and IgnisHPC considering all the Big Data applications detailed above. Speedups were calculated using as reference the Spark sequential time. All experiments have been executed ten times and their average result is reported. In addition, the relative standard deviation (RSD), also known as the coefficient of variation, is calculated as $100 \times \sigma / \bar{x}$. Low RSD values point out that the data is tightly clustered around the mean.

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



(a) Times (strong scaling) (b) Speedup

Figure 16: Study of the scalability of IgnisHPC and Apache Spark running the TeraSort application.



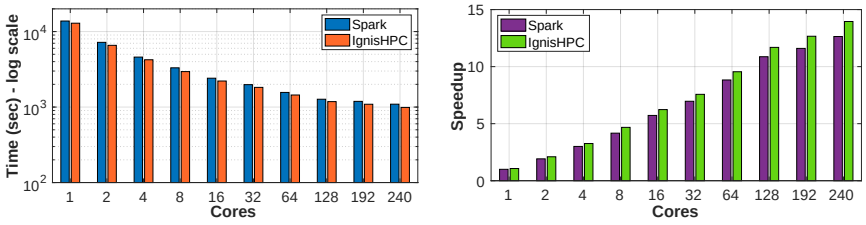
(a) Times (strong scaling) (b) Speedup

Figure 17: Study of the scalability of IgnisHPC, Ignis and Apache Spark running the K-Means application.

Figures 14 and 15 show the scalability study of the *Minebench* (MB) application using two different implementations. In the first one, MB was programmed using only Python, while in the second one, the two chained map operations are implemented using Python (data-intensive task) and C++ (compute-intensive task), respectively. Results show that IgnisHPC exhibits very good strong-scaling behavior for both implementations. On the contrary, the Spark scalability is impacted for the cost of starting JVMs and transferring data through system pipes to the Python processes, causing an important degradation in the overall performance [8]. This scenario is even more clear for the multi-language implementation in Figure 15(a) since Spark sends data from Python to C++ processes through the JVM, increasing the number of pipe operations. As a consequence, the Spark strong scalability is really poor. On the other hand, IgnisHPC weak scales very well for both code versions (Figures 14(c) and 15(c)), which is not surprising since there is not much communication in the MB application. As a consequence, IgnisHPC is able to extract all the existent parallelism. Finally, we must highlight that IgnisHPC clearly outperforms Ignis both in terms of strong and weak scalability, especially when considering the multi-language application. The RSD for the Minebench experiments ranges from 0.3% to 4.9%. Note that for the Python implementation (Figure 14), the performance differences between IgnisHPC and Ignis are only caused by architectural improvements in the framework since no MPI operations are carried out. Executors read the input data from a file and exchange partial results using the shared memory. However, if we consider the multi-language implementation of MB (Figure 15), performance improvements are also due to the use of MPI for the communication between *Workers*.

Performance results of the TeraSort (TS) application running on IgnisHPC and Spark frameworks are displayed in Figure 16. Ignis results are not shown because the memory consumption of sorting 1 TB of data is too high for our cluster. As we explained in Section 3.8, Ignis assigns one data partition to each executor. For TS those partitions are very large. Every time an element is added to a partition, the complete partition is copied to a different memory location (realloc operation), which causes a boost in the memory requirements. This restricts Ignis to work with smaller input datasets. We avoid this limitation since IgnisHPC was designed to allow several partitions per worker. Two different TS implementations in IgnisHPC were analyzed: a pure Python code and a multi-language Python-C++ code. In the latter case, the sort operation uses a user-defined C++ function for comparison purposes. For all the cases IgnisHPC outperforms Spark, especially when considering the multi-language implementation. In this way, for instance, IgnisHPC is 116× faster than sequential Spark when using 160 cores, while Spark reaches a speedup of only 66× (Figure 16(b)). It allows IgnisHPC to sort 1 TB of data in barely 5 minutes. The RSD for the TS experiments ranges from 1.3% to 6%.

Strong scaling results of K-Means (KM) are shown in Figure 17. We used as reference the Spark implementation of this algorithm included in MLlib (Machine Learning Library) [30]. For Ignis, a pure C++ and a Python-C++ implementations of KM were analyzed. For IgnisHPC, we only show the results for the multi-language Python-C++ code because the numbers obtained by a pure C++ application are very similar. We can observe that Ignis was able to beat Spark when considering the C++ code. However, an important degradation in the scalability was detected for the multi-language implementation as the parallelism increases. This problem was explained in Section 3.6



(a) Times (strong scaling) (b) Speedup

Figure 18: Study of the scalability of IgnisHPC and Apache Spark running the PageRank application.

and is related to the way Ignis handles iterative applications. Ignis starts and stops the executors each iteration because the driver must compute the partial results, which has an important impact on the performance. To deal with this, IgnisHPC takes advantage of MPI in such a way that executors compute the partial results and share them without intervention of the driver. For this reason IgnisHPC exhibits a very good strong scalability even for multi-language iterative applications, decreasing noticeably the execution times with respect to Spark and Ignis. It can be observed in Figure 17(b) that IgnisHPC is about two and four times faster than Spark and Ignis (multi-language code) when using all the cores in the cluster, respectively. The RSD for the KM experiments ranges from 1.1% to 4.8%.

In Big Data analytics many problems require processing graphs. For this reason it is essential for a Big Data framework as IgnisHPC to include primitives to support this kind of applications. In addition, since the size of the graphs to be processed is often very large, a good scalability is essential. With this goal in mind we have evaluated two well-known graph algorithms in Spark and IgnisHPC: PageRank (PR) and Transitive Closure (TC). Performance results are shown in Figures 18 and 19, respectively. The algorithms in Spark were implemented using GraphX [31]. Note that Ignis does not support several operations that are basic for this type of applications such as `join` and `union` (see Table 3), so it cannot be evaluated. Despite the fact that GraphX is a highly tuned API for graph processing, IgnisHPC is capable of outperforming Spark in both cases. The RSD for the PR experiments ranges from 0.6% to 2.7%, while for the TC varies from 0.1% to 1.7%.

6.2.2. Final remarks

Table 3 summarizes the performance gains obtained by IgnisHPC with respect to Spark and Ignis when running all the considered Big Data applications. Results were obtained using the maximum number of cores available and taking into account the best Ignis and Spark implementation (if there is more than one). In this way, for example, two implementations are available for Minebench, pure Python and multi-language Python-C++. In that case we used as reference for Spark the Python code, while for Ignis the best performing implementation was the multi-language one (see the values in Figures 14(a) and 15(a) when using 240 cores).

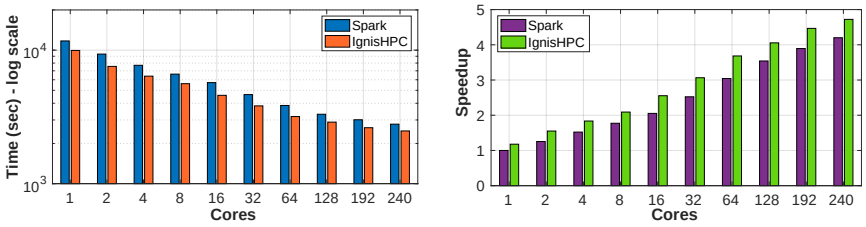
According to the results, IgnisHPC is from 1.10× to 3.87× faster than Spark. The good behavior of IgnisHPC is particularly relevant when considering multi-language applications. At the same time, IgnisHPC is a step forward with respect to Ignis in terms of performance. In this case, IgnisHPC is from 1.08× to 1.28× faster than Ignis. However, there are additional benefits. First, the memory consumption in IgnisHPC was optimized allowing multiple partitions per executor, which allows to work with extremely large datasets. That is the reason why Ignis is not able to execute TeraSort in our cluster. And second, the IgnisHPC API was extended to support, among others, graph processing algorithms such as PageRank and Transitive Closure.

6.3. HPC applications

For many years MPI has been the dominant parallel programming model in the HPC area. As we explained in Section 5, thanks to its architectural design, one of the most important features of IgnisHPC is its ability to execute native MPI applications within the framework just adding a few lines of code. To evaluate the benefits of our approach we are interested in two key areas: performance (with respect to the native MPI execution) and productivity (additional Source Lines Of Code - SLOC). In this way, we have selected five HPC applications coming from different scientific fields that represent a variety of MPI communication patterns. Table 4 summarizes the most important MPI calls (point-to-point and collective operations) used in the applications. Note that during a specific run, an application may use only a subset of these communications. All the codes were implemented using C/C++. For some of them we have considered hybrid implementations (MPI+OpenMP) to demonstrate that is also possible to efficiently execute this type of applications in IgnisHPC without additional effort.

Next we provide some information about the selected HPC applications used in the experimental evaluation:

– *LULESH (Livermore Unstructured Lagrange Explicit Shock Hydrodynamics).* It is a shock hydrodynamics code developed at Lawrence Livermore National Lab (LLNL) [22]. It has been ported to a number of programming models: MPI, OpenMP, MPI+OpenMP, CUDA, etc. In this paper we have considered the hybrid MPI+OpenMP implementation, which uses MPI between nodes and OpenMP for cores on a node. Performance tests were run on 8 nodes with a problem size of 70^3 on each node, which corresponds to the most representative problem size [32].



(a) Times (strong scaling)

(b) Speedup

Figure 19: Study of the scalability of IgnisHPC and Apache Spark running the Transitive Closure application.

Application	No. times faster than Spark	No. times faster than Ignis
Minebench	3.87× [Python & C++] 1.26× [Python]	1.23× [Python & C++] 1.08× [Python]
TeraSort	1.76× [C++] 1.35× [Python]	–
K-Means	1.94× [Python & C++]	1.28× [Python & C++]
PageRank	1.10× [Python]	–
Transitive Closure	1.12× [Python]	–

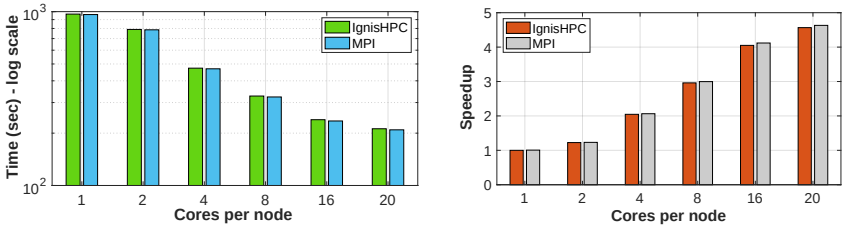
Table 3: Summary of the IgnisHPC performance results for all the Big Data applications considering the maximum number of cores and the best Ignis and Spark implementation (in case there is more than one). Between brackets the programming language/s used in the IgnisHPC implementation.

Application	Point-to-point		Collective	
	Blocking	Non-blocking	Blocking	Non-blocking
LULESH	–	`Isend`, `Irecv`	`Allreduce`, `Barrier`	–
AMG	`Send`, `Recv`	`Isend`, `Irecv`, `Irsend`	`Allreduce`, `Barrier`, `Bcast`, `Reduce`, `Alltoall`, `Allgather(v)`, `Gather(v)`, `Scan`, `Scatter(v)`	–
MiniAMR	`Send`, `Recv`	`Isend`, `Irecv`	`Allreduce`, `Barrier`, `Bcast`, `Alltoall`	–
MiniVite	`Sendrecv`	`Isend`, `Irecv`	`Allreduce`, `Barrier`, `Bcast`, `Reduce`, `Alltoall(v)`, `Exscan`	`Ialltoall`
MSAProbs	`Send`, `Recv`	`Isend`, `Irecv`	`Allreduce`, `Barrier`, `Bcast`	–

Table 4: MPI calls used for communications in the HPC applications.

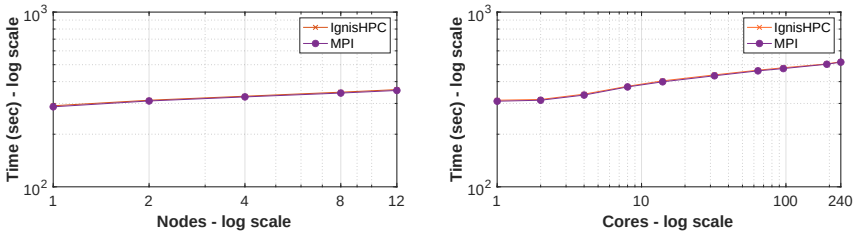
– *AMG*. It is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. It is part of the Exascale Computing Project (ECP) proxy applications suite[5] and was derived directly from the BoomerAMG [33] solver. AMG is an SPMD application with about 65,000 lines of code which uses OpenMP threading within MPI tasks. Parallelism is achieved by simply subdividing the grid into logical $P \times Q \times R$ (in 3D) chunks of equal size. AMG is a highly synchronous and memory-access bound code. The scalability tests were obtained with a fixed local problem grid size per MPI process of 100×100×100 points.
– *miniAMR*. It is a proxy app for adaptive mesh refinement (AMR), which is a frequently used technique for efficiently solving partial differential equations (PDEs) [34]. It applies a stencil calculation on a unit cube computational domain, which is divided into blocks. This application also belongs to the ECP proxy app collection and was implemented using MPI. We used blocks with dimensions 8×8×8 and a maximum of 4 refinement levels. The test case we considered is that of an expanding sphere, which closely mimics an explosion. Blocks are refined along the boundary of the expanding sphere.
– *miniVite*. It implements a parallel Louvain method for community detection, which is one of the most important graph kernels used in scientific and social networking applications for discovering higher order structures within a graph [35]. It is also included in ECP proxy app collection. miniVite was programmed using MPI and OpenMP. As input we used a graph with 10% of the vertices of the well-known *friendster* social network graph [36]. It consists of 6.6M vertices and 24.2M edges.
– *MSAProbs*. One basic step in many bioinformatics analyses is the multiple sequence alignment (MSA). MSAProbs [37] is a state-of-the-art tool to compute protein MSA based on hidden Markov models. In this work we have considered its MPI+OpenMP parallel implementation [38]. The input dataset *PF*07085 [39] used in the tests consists of 975 sequences with an average length of 512.

[5] http://proxyapps.exascaleproject.org/ecp-proxy-apps-suite



(a) Times (strong scaling) (b) Speedup

Figure 20: Study of the scalability of LULESH (8 nodes).



(a) (b)

Figure 21: Study of the weak scalability of AMG (20 threads/cores per node) (a) and miniAMR (b).

6.3.1. Analysis and discussion

Next we carry out the analysis of the execution of the MPI-based HPC applications within IgnisHPC. As mentioned previously, we will focus on two aspects. First, the performance differences between running the HPC applications on the cluster as native MPI tasks or using IgnisHPC. It is important to highlight that is out of the scope of this paper to analyze the particular behavior of each MPI application in terms of performance and scalability. This was extensively explained in the references provided in the description paragraphs of Section 6.3. The second key aspect is productivity. In our case we measured the source lines of code (SLOC) of the applications. This metric is very important since scientists will only adopt IgnisHPC to execute MPI applications if porting them requires little effort.

Performance. Figure 20 shows the strong scaling results of LULESH. Note that this application uses a hybrid MPI+OpenMP approach. Performance differences between native MPI and IgnisHPC are really small, always lower than 1.7%. Running LULESH from IgnisHPC shows the same scalability trend than the MPI native execution.

Weak scalability tests were run to evaluate AMG (MPI + OpenMP) and miniAMR (MPI). Results are shown in Figures 21(a) and 21(b). In both cases also, the performance of IgnisHPC comes close to that of its counterpart, the native MPI implementation. The maximum performance difference is only about 1.4% for both applications. In this way, for instance, the execution times of miniAMR using all the cores in the cluster were 520 and 517 seconds with native MPI and IgnisHPC, respectively.

miniVite scalability results are displayed in Figure 22. The behavior replicates the observations commented previously for the other HPC applications. That is, running an MPI application within the IgnisHPC framework achieves very similar performance with respect to the native execution. In this particular case, the maximum difference drops to only 0.2%. The same scalability analysis applied to MSAProbs (Figure 23) produces a maximum performance difference of 0.4%.

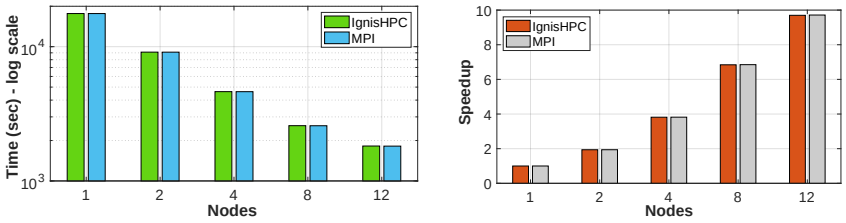
So we conclude that running MPI (and MPI+OpenMP) applications from IgnisHPC is as efficient as executing them natively.

Productivity. As we explained in Section 5, running MPI applications in IgnisHPC requires some minimal modifications to the original source code and adding a few lines to call the application from the driver code. It is important to highlight that most of these extra lines are devoted to parsing the arguments of the MPI application. In any case, this is a very simple and repetitive code as shown in the example of Figure 10, which can be considered as *boilerplate*. We measure the SLOC using SLOCcount [40] of each original MPI application and its counterpart adapted to IgnisHPC (see Table 5). The number of extra lines, between brackets in the table, ranges only from 17 to 75. This demonstrates that integrating MPI applications and libraries in IgnisHPC is a straightforward process, which is very important for the HPC community since it is not necessary to port MPI codes to a new API or programming model. Therefore, IgnisHPC fulfills its design goal of unifying in a single framework the benefits of HPC and Big Data applications.

7. Related Work

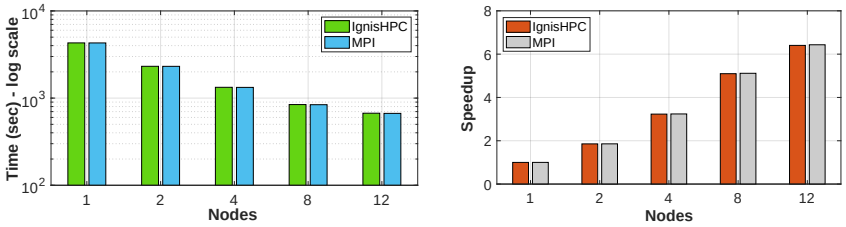
7.1. HPC and Containers

HPC workloads tend to be monolithic in nature so that each component and its dependencies must be present for running or compiling the code. In addition, if an update of any component



(a) Times (strong scaling) (b) Speedup

Figure 22: Study of the scalability of miniVite (20 threads/cores per node).



(a) Times (strong scaling) (b) Speedup

Figure 23: Study of the scalability of MSAProbs (20 threads/cores per node).

Application	**SLOC MPI**	**SLOC** IgnisHPC
LULESH	5,918	5,993 (+75)
AMG	65,154	65,197 (+43)
MiniAMR	9,958	9,987 (+39)
MiniVite	3,264	3,324 (+60)
MSAProbs	6,045	6,062 (+17)

Table 5: SLOC of the HPC applications.

is required, all modules will be affected. For this reason, the most common difficulty faced by end-users when creating and implementing scientific software is the installation and configuration of a framework with thousands of dependencies.

Containers are a good way to self-contain an application and its dependencies in a controlled environment. Containers do not interfere with each other and allow to be deleted or updated without leaving any trace on the physical machine. They are an alternative to virtual machines while maintaining a similar level of isolation and showing a superior performance that in some cases is almost identical to the one obtained when executing natively on a real machine [41, 42].

IgnisHPC can be seen as an MPI application, so it can be efficiently executed inside containers as it was proven in several works. For example, running MPI applications on a containerized cluster using Docker on a cluster [43] or in the Cloud [44], or using Shifter [45] instead. Other works deal with the orchestration of Docker containers in an HPC environment. For instance, Higgins et al. [46] implemented a script based on SSH for the creation of an MPI environment inside a Docker container. Unlike IgnisHPC, this approach requires root privileges to modify the hosts configuration. Another paper introduces Scylla [47], a framework for deploying MPI jobs within Docker Containers using Apache Mesos. As explained in Section 2, Apache Mesos requires an orchestration framework such as Marathon or Singularity to be used. However, the authors, instead of considering a third party framework, implemented an ad-hoc solution. Their approach also requires root privileges. We must highlight that IgnisHPC supports all the functionalities included in Scylla without needing root permissions and is not limited to work with Apache Mesos.

7.2. Spark and HPC applications

As a general-purpose framework, Spark has been widely used for many scientific applications and algorithms. However, there are examples from different areas such as linear algebra [48], genomics [49] or even data science [50] where Spark does not obtain the expected performance.

One way to approach Big Data and HPC worlds is trying to boost the performance of well-established Big Data technologies when running on HPC systems. For example, taking advantage of the Infiniband fast interconnection network [51] or the standard HPC programming models such as MPI. We are especially interested in those works that opt for the latter approach. For instance, Anderson et al. [52] try to combine Spark and MPI. They offload computations to an MPI environment from within Spark in such a way that Spark and MPI tasks run at the same time, using a socket-based implementation for efficient data exchange between processes. In their approach the results of the MPI processing are copied back to persistent storage (HDFS), and then into Spark for further processing. As a consequence, for those applications that require few iterations and/or less work per iteration, there is a degradation in the performance. However, their approach shows a good behavior with several graph and machine learning applications that do not require a lot of data movement between Spark and MPI environments.

A similar solution can be found in [53]. They introduce Alchemist, a TCP socket-based implementation for inter-process communication between Spark and MPI. Alchemist was designed to call MPI-based libraries from Spark using the Scala programming language. Like previous work, due to the use of two type of tasks for MPI and Spark, it is always necessary to keep two copies of the same data. In addition, moving data between Spark and MPI processes is costly since TCP sockets are often a slower alternative with respect to shared memory. Therefore, this approach is also limited to computationally-intensive applications for which the cost associated with data transfers is negligible when compared to the overhead that would have been incurred by Spark.

Finally, there are several related works of the same research group that make a better integration between Spark and HPC technologies. In [54], Spark and MPI tasks share the same process, which removes the overhead caused by the data transfers. Python is used as programming language since Spark and MPI has an interface for this language. However, as we explained in [8], Python is not natively supported by Spark which causes an important degradation in its overall performance. IgnisHPC overcomes that limitation using native executors for each supported programming language.

In their next works, the authors introduce Spark-MPI [55, 56], a hybrid platform that combines Spark and MPI taking advantage of the MPI Exascale Process Management Interface (PMIx). A Spark-MPI application consists of a driver launched by Spark and a set of processes launched by MPI. These MPI processes connect to the Spark driver as workers, and they will be able to execute both RDD and MPI functions. Note that using MPI routines in Spark-MPI only makes sense when the data is processed using `mapPartitions`. That is the only way that Spark provides to work on a complete partition instead of on each element of the partition. The authors showed the benefits of Spark-MPI with deep learning algorithms and ptychographic and tomographic applications. However, Spark-MPI has several limitations that we detailed next:

- Spark-MPI requires a hybrid environment for Spark and MPI, which will be configured with their respective resource managers. For example, Mesos or Yarn for Spark and Hydra or Slurm for MPI. It is hard to find a system configured this way since resource managers cannot share the available hardware resources. Moreover, a Spark-MPI job should queue for both Spark and MPI queuing systems, and the requested resources may not be granted at the same time. On the other hand, IgnisHPC is a single application so the previous problems do not apply.
- Due to its particular architecture and how Spark executors are launched, Spark-MPI loses the fault tolerance system provided by Spark. As a consequence, after any failure in the executors, all the job is lost. On the contrary, as we demonstrated in [8], Ignis and IgnisHPC are able to recover after a failure of a cluster node or some of the executors. If some data is lost, IgnisHPC has enough information about how it was derived in such a way that only those operations needed to recompute the corresponding portion of data are performed.
- Spark-MPI can only be used with Python codes (PySpark). Although communications are performed using MPI, Spark-MPI would experience the same degradation in the performance observed for Spark when executing Python applications (see Section 6.2.1). Spark suffers performance issues since it requires sharing data outside the JVM through system pipes. Therefore, Spark-MPI performance results would be similar to those obtained by Spark when running Python codes (see, for example, the Minebench results in Figure 14). On the other hand, although the execution of pure MPI codes in Spark-MPI is discussed, the vast majority of MPI applications are implemented in C/C++ for which Spark and Spark-MPI does not have native support.
- Spark-MPI is limited to access one partition at the same time per executor. Partition size is restricted to a maximum of 2 GB, which is related to the use of JVMs in Spark. Therefore, additional executors should be created in case more data is necessary, degrading the overall I/O performance. This restriction only applies to IgnisHPC for Java applications, but not for Python and C/C++.

8. Conclusions

In this work we have introduced a new computing framework named IgnisHPC[6] to fill the gap between Big Data and HPC languages and programming models. IgnisHPC supports the combination of JVM and non-JVM-based languages in the same application (currently, Java, Python and C/C++). It was designed to take advantage of MPI for communications, which allows the framework to execute efficiently MPI applications and libraries. As a consequence, the MPI-based HPC scientific applications do not have to be ported to a new API or programming model. Moreover, it is possible to combine in the same multi-language code HPC tasks (using MPI) with Big Data tasks (using MapReduce operations).

The experimental evaluation demonstrated the benefits of our proposal in terms of performance with respect to the *de-facto* standard for Big Data processing, Spark, and our first prototype of multi-language framework, Ignis. In particular, IgnisHPC is from 1.1× to 3.9× faster than Spark, and about 1.2× faster than Ignis. In the same way, we observed that running MPI and MPI+OpenMP applications in IgnisHPC is as efficient as executing them natively. Therefore, thanks to IgnisHPC we are merging both Big Data and HPC software ecosystems in just one execution environment.

References

[1] S. Heldens, *et al.*, The Landscape of Exascale Research: A Data-Driven Literature Analysis, ACM Comput. Surv. 53 (2) (2020).

[2] T. White, Hadoop: The Definitive Guide, 4th Edition, O'Reilly Media, Inc., 2015.

[6] It is publicly available at `https://github.com/ignishpc`

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster Computing with Working Sets, in: Proc. of the 2nd USENIX Conf. on Hot Topics in Cloud Computing (HotCloud), 2010, pp. 10–10.

[4] M. Asch, *et al.*, Big data and extreme-scale computing: Pathways to Convergence-Toward a shaping strategy for a future software and data ecosystem for scientific inquiry, IJHPCA 32 (4) (2018) 435–479.

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

[6] M. Ding, *et al.*, More Convenient More Overhead: The Performance Evaluation of Hadoop Streaming, in: Proc. of the ACM Symposium on Research in Applied Computation, 2011, pp. 307–313.

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

[8] C. Piñeiro, R. Martínez-Castaño, J. C. Pichel, Ignis: An efficient and scalable multi-language Big Data framework, Future Generation Computer Systems 105 (2020) 705–716.

[9] B. Alverson, E. Froese, L. Kaplan, D. Roweth, Cray XC series network, Cray Inc., White Paper WP-Aries01-1112 (2012).

[10] D. De Sensi, S. Di Girolamo, K. H. McMahon, D. Roweth, T. Hoefler, An In-Depth Analysis of the Slingshot Interconnect, in: Proceedings of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC), 2020.

[11] Y. Ajima, *et al.*, The Tofu Interconnect D, in: IEEE Int. Conference on Cluster Computing (CLUSTER), 2018, pp. 646–654.

[12] MPICH, `https://www.mpich.org`, [Online; accessed October, 2021].

[13] Open-MPI, `https://www.open-mpi.org/`, [Online; accessed October, 2021].

[14] B. Hindman, *et al.*, Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center, in: Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation, 2011, p. 295–308.

[15] HashiCorp, Nomad: workload orchestration made easy, `https://www.nomadproject.io/`, [Online; accessed October, 2021].

[16] D. Merkel, Docker: lightweight linux containers for consistent development and deployment, Linux journal 2014 (239) (2014) 2.

[17] Apache Marathon, `https://mesosphere.github.io/marathon/`.

[18] Apache Singularity, `https://getsingularity.com/`.

[19] J. T. Kukunas, V. Gopal, J. Guilford, S. Gulley, A. van de Ven, W. Feghali, High Performance ZLIB Compression on Intel Architecture Processors, Tech. rep., Intel (2014).

[20] M. Zaharia, *et al.*, Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing, in: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, 2012, pp. 2–2.

[21] M. de Bayser, R. Cerqueira, Integrating MPI with Docker for HPC, in: IEEE Int. Conference on Cloud Engineering (IC2E), 2017, pp. 259–265.

[22] I. Karlin, *et al.*, Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application, in: 27th Int. Symposium on Parallel and Distributed Processing, 2013, pp. 919–932.

[23] V. K. Vavilapalli, *et al.*, Apache Hadoop YARN: Yet Another Resource Negotiator, in: Proc. of the 4th Annual Symposium on Cloud Computing, ACM, 2013, pp. 5:1–5:16.

[24] J. Shi, *et al.*, Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics, Proc. VLDB Endowment 8 (13) (2015) 2110–2121.

[25] S. Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System (2008). URL `http://bitcoin.org/bitcoin.pdf`

[26] R. C. Merkle, Protocols for public key cryptosystems, in: IEEE Symposium on Security and Privacy, 1980, pp. 122–122.

[27] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, H. Shi, On the Versatility of Parallel Sorting by Regular Sampling, Parallel Computing 19 (1993) 1079–1103.

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

[29] J. Leskovec, A. Krevl, SNAP Datasets: Stanford Large Network Dataset Collection, `http://snap.stanford.edu/data` (2014).

[30] X. Meng, *et al.*, MLlib: Machine Learning in Apache Spark, The Journal of Machine Learning Research 17 (1) (2016) 1235–1241.

[31] R. S. Xin, J. E. Gonzalez, M. J. Franklin, I. Stoica, GraphX: A Resilient Distributed Graph System on Spark, in: 1st International Workshop on Graph Data Management Experiences and Systems, ACM, 2013.

[32] I. Karlin, J. McGraw, J. Keasler, B. Still, Tuning the LULESH Mini-app for Current and Future Hardware, Tech. rep. (2013).

[33] V. E. Henson, U. M. Yang, BoomerAMG: A parallel algebraic multigrid solver and preconditioner, Applied Numerical Mathematics 41 (1) (2002) 155–177.

[34] A. Sasidharan, M. Snir, MiniAMR - A miniapp for Adaptive Mesh Refinement, Tech. rep. (2016).

[35] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, A. H. Gebremedhin, MiniVite: A Graph Analytics Benchmarking Tool for Massively Parallel Systems, in: IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), 2018, pp. 51–56.

[36] J. Yang, J. Leskovec, Defining and Evaluating Network Communities based on Ground-truth (2012). arXiv:1205.6233.

[37] Y. Liu, B. Schmidt, D. L. Maskell, MSAProbs: multiple sequence alignment based on pair hidden Markov models and partition function posterior probabilities, Bioinformatics 26 (16) (2010) 1958–1964.

[38] J. González-Domínguez, Y. Liu, J. Touriño, B. Schmidt, MSAProbs-MPI: parallel multiple sequence aligner for distributed-memory systems, Bioinformatics 32 (24) (2016) 3826–3828.

[39] J. Mistry, *et al.*, Pfam: The protein families database in 2021, Nucleic Acids Research 49 (D1) (2020) D412–D419.

[40] D. Wheeler, SLOCCount, `http://www.dwheeler.com/sloccount`, [Online; accessed November, 2021].

[41] T. Adufu, J. Choi, Y. Kim, Is container-based technology a winner for high performance scientific applications?, in: 17th Asia-Pacific Network Operations and Management Symp. (APNOMS), 2015, pp. 507–510.

[42] M. T. Chung, N. Quang-Hung, M.-T. Nguyen, N. Thoai, Using Docker in high performance computing applications, in: IEEE 6th Int. Conference on Communications and Electronics (ICCE), 2016, pp. 52–57.

[43] L. Benedicic, F. A. Cruz, A. Madonna, K. Mariotti, Portable, high-performance containers for HPC (2017). arXiv:1704.03383.

[44] A. J. Younge, K. Pedretti, R. E. Grant, R. Brightwell, A Tale of Two Systems: Using Containers to Deploy HPC Applications on Supercomputers and Clouds, in: IEEE Int. Conference on Cloud Computing Technology and Science (CloudCom), 2017, pp. 74–81.

[45] P. Saha, A. Beltre, P. Uminski, M. Govindaraju, Evaluation of Docker Containers for Scientific Workloads in the Cloud, in: Proc. of the Practice and Experience on Advanced Research Computing, 2018.

[46] J. Higgins, V. Holmes, C. Venters, Orchestrating Docker Containers in the HPC Environment, in: HPC. Lecture Notes in Computer Science, vol 9137., Springer Int. Publishing, 2015, pp. 506–513.

[47] P. Saha, A. Beltre, M. Govindaraju, Scylla: a Mesos Framework for Container Based MPI Jobs, CoRR abs/1905.08386 (2019). arXiv:1905.08386.

[48] A. Gittens, *et al.*, Matrix factorizations at scale: A comparison of scientific data analytics in Spark and C+MPI using three case studies, in: IEEE Int. Conf. on Big Data, 2016, pp. 204–213.

[49] J. M. Abuín, N. Lopes, L. Ferreira, T. F. Pena, B. Schmidt, Big Data in metagenomics: Apache Spark vs MPI, Plos One 15 (10) (2020) 1–20.

[50] M. Saxena, S. Jha, S. Khan, J. Rodgers, P. Lindner, E. Gabriel, Comparison of MPI and Spark for Data Science Applications, in: IEEE Int. Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2020, pp. 682–690.

[51] X. Lu, *et al.*, High-Performance Design of Hadoop RPC with RDMA over InfiniBand, in: 42nd Int. Conference on Parallel Processing, 2013, pp. 641–650.

[52] M. Anderson, *et al.*, Bridging the Gap between HPC and Big Data Frameworks, Proc. VLDB Endow. 10 (8) (2017) 901–912.

[53] A. Gittens, *et al.*, Accelerating Large-Scale Data Analysis by Offloading to High-Performance Computing Libraries Using Alchemist, in: Proc. of the 24th ACM SIGKDD Int. Conference on Knowledge Discovery & Data Mining, 2018, p. 293–301.

[54] N. Malitsky, Bringing the HPC reconstruction algorithms to Big Data platforms, in: NY Scientific Data Summit (NYSDS), 2016, pp. 1–8.

[55] N. Malitsky, *et al.*, Building near-real-time processing pipelines with the Spark-MPI platform, in: NY Scientific Data Summit (NYSDS), 2017, pp. 1–8.

[56] N. Malitsky, R. Castain, M. Cowan, Spark-MPI: Approaching the Fifth Paradigm of Cognitive Applications (2018). arXiv:1806.01110.