IgnisHPC

Release 2.0

César Piñeiro and Juan C. Pichel

Jul 13, 2022

Contents

About

One of the most important issues in the path to the convergence of High Performance Computing (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 (currently Java, Python and C/C++). 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 a 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 such as Spark. *IgnisHPC* is publicly available for the Big Data and HPC research community.

1 Getting Started

IgnisHPC is a modularized docker framework consisting of multiple source code repositories. The framework is open source, so all repositories can be found in GitHub: IgnisHPC.

Next we summarize the minimum steps to execute a simple job in IgnisHPC.

1.1 Requirements

As we mentioned, IgnisHPC is a dockerized framework, so all the system modules are executed inside docker containers. On the other hand, IgnisHPC external dependencies such as schedulers or image storage can be installed independently, but IgnisHPC includes a dockerized version of them.

Therefore, the minimum requirements to run Ignis are:

1. *Docker*: It must be installed and accessible. We recommend using the newest version available.
2. *Python3*: Available by default in most Linux distributions. It is used to execute a deploy script and simplify the installation of IgnisHPC and its dependencies.
3. *Pip*: The deploy script is available as a pip package, although it can be downloaded from the source code repository. In any case, using pip is the easiest way to install IgnisHPC.
4. *Git* (optional): The git binary is required for building IgnisHPC images from repositories.

1.2 Installation

IgnisHPC can be installed just using the following command:

```
$ pip install ignishpc
```

Once the command is executed, we can check that we have the `ignis-deploy` command available in the path. This process must be carried out in all the computing nodes where IgnisHPC is going to be executed.

1.3 Creating IgnisHPC Containers

Images of IgnisHPC containers are not available for download. They must be built in your runtime environment. This allows the creation of a custom development environment with complete isolation from other framework installations even on the same machines.

In this example we will use a local docker repository and ignishpc as the base path for all images.

```
$ ignis-deploy registry start --default
```

This command will launch a docker repository that will be available on port 5000.
`--default` parameter indicates that other calls to `ignis-deploy` should use this repository as the default source.

Once executed, we will receive the following warning message:

```
info: add '{insecure-registries" : [ "myhost:5000" ]}' to /etc/docker/daemon.json and␣
↪restart
       docker daemon service
      use myhost:5000 to refer the registry
```

The docker registry requires a certificate for validation. We can create one locally or add `{insecure-registries" : [ "myhost:5000" ]}` to `/etc/docker/daemon.json` with the aim of forcing docker to use an insecure one. Do not forget to restart the docker daemon to reload the configuration.

Once the registration is available, we can proceed with the creation of the IgnisHPC images:

```
$ ignis-deploy images build --full --sources \
   https://github.com/ignishpc/dockerfiles.git \
   https://github.com/ignishpc/backend.git \
   https://github.com/ignishpc/core-cpp.git \
   https://github.com/ignishpc/core-python.git \
       https://github.com/ignishpc/core-go.git
```

The first two repositories are essential for the construction of the base images. The command can be executed in several phases, it is not necessary to specify all the repositories in the same execution. The only restriction is that if you use the `--full` parameter, which creates an extra image with all the core repositories, you must have all the cores together. This allows users to create an image that can run Python, C++ and Go codes in the same container.

Finally, IgnisHPC files can be extracted from an image with:

```
$ docker run --rm -v $(pwd):/target <ignis-image> ignis-export-all /target
```

The result will not be executable but can be used in application development.

1.4 Deploying IgnisHPC Containers

Once all images are created, it is necessary to deploy the containers. IgnisHPC jobs are launched using the submitter module, but to use a cluster it requires a resource and scheduler manager such as Nomad or Mesos.

Alternatively, IgnisHPC can also be launched as a slurm job in an HPC cluster, docker is replaced by singularity, so a different submitter must be used.

Next we show examples deploying the containers locally (no manager is needed), and using Nomad, Mesos and Slurm.

Docker (Only local)

Submitter using an http endpoint:

```
$ ignis-deploy submitter start --dfs <working-directory-path> --scheduler docker tcp://
↪myhost:2375
```

Submitter using a Unix-socket:

```
$ ignis-deploy submitter start --dfs <working-directory-path> --scheduler docker /var/
↪run/docker.sock \
  --mount /var/run/docker.sock /var/run/docker.sock
```

Nomad

Master node:

```
$ ignis-deploy nomad start --password 1234 --volumes <working-directory-path\*>
```

Worker nodes:

```
$ ignis-deploy nomad start --password 1234 --join myhost1 --default-registry myhost1:5000
```

Submitter:

```
$ ignis-deploy submitter start --dfs <working-directory-path\*> \
   --scheduler nomad http://myhostX:4646
```

* The working directory must be available on all nodes via NFS (Network File System) or a DFS (Distributed File System). (Only required for working with files)

Mesos

Zookeeper is requiered by Mesos:

```
$ ignis-deploy zookeeper start --password 1234
```

Master node:

```
$ ignis-deploy mesos start -q 1 --name master -zk  zk://master:2281 \
   --service [marathon | singularity] --port-service 8888
```

Worker nodes:

```
$ ignis-deploy mesos start --name nodoX -zk  zk://master:2281 \
   --port-service 8888 --default-registry master:5000
```

Submitter:

```
$ ignis-deploy submitter start --dfs <working-directory-path*> \
   --scheduler [marathon | singularity] http://master:8888
```

* The working directory must be available on all nodes via NFS (Network File System) or a DFS (Distributed File System). (Only required for working with files)

Slurm

The `ignis-slurm` submitter can be obtained from `ignishpc/slurm-submitter` with:

```
$ docker run --rm -v $(pwd):/target ignishpc/slurm-submitter ignis-export /target
```

This submitter will allow you to launch ignisHPC on a cluster as a non-root user and without docker.

IgnisHPC Docker images can be converted to singulairty image files with:

```
$ ignis-deploy images singularity [--host] ignishpc/full ignis_full.sif
```

The basic syntax of `ignis-slurm` is the same as the later shown `ignis-submit`, but a first parameter with job-time must be passed to be requested to slurm. The time can be specified in any format supported by slurm. For example, a 10 minute job should start with:

```
$ ignis-slurm 00:10:00 ....
```

In addition, help text can be displayed using:

```
$ ignis-slurm --help
```

1.5 Launching the first job

The first step to launch a job is to connect to the submiter container. The default password is `ignis`, but we can change it inside the container or choose one when launching the submitter.:

```
$  ssh root@myhost -p 2222
```

The code we will use as an example is the classic Wordcount application, which can be seen below.

```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/python"
prop["ignis.executor.instances"] = "1"
prop["ignis.executor.cores"] = "2"
prop["ignis.executor.memory"] = "1GB"
# Construction of the cluster
cluster = ignis.ICluster(prop)

# Initialization of a Python Worker in the cluster
worker = ignis.IWorker(cluster, "python")
# Task 1 - Tokenize text into pairs ('word', 1)
text =  worker.textFile("text.txt")
words = text.flatmap(lambda line: [(word, 1) for word in line.split()])
# Task 2 - Reduce pairs with same word and obtain totals
count = words.toPair().reduceByKey(lambda a, b: a + b)
# Print results to file
count.saveAsTextFile("wordcount.txt")

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

In order to run it, we need to create a file containing a text sample (`text.txt`) and store it in the working directory. By default the submitter sets the working directory to `/media/dfs`. All relative paths used in the source code are resolved using this working directory, so `/media/dfs/text.txt` is an alias of `text.txt`.

Finally, we can execute our code using the submitter:

```
$ ignis-submit ignishpc/python python3 driver.py
```

or:

```
$ ignis-submit ignishpc/python ./driver.py
```

When the execution has finished, we can see the result of the execution in `wordcount.txt` located in the working directory. If we want to check the execution logs, we must navigate to the scheduler web or use `docker log` in case of using docker directly.

Launching without Container

The `ignis-submit` can also be used outside the submiter container, for example where permanent containers are not allowed:

```
$ docker run --rm -v $(pwd):/target ignishpc/submitter ignis-export /target
```

This command will create a `ignis` folder in the current directory with everything needed to run the submiter. The `ignis-deploy` command configures the submitter container, but when there is no container, we must set the configuration manually. The submitter needs a dfs and a scheduler, as `ignis-deploy` showed, these can be defined as environment variables or in `ignis/etc/ignis.conf` property file.

```
# set current directory as job directory (ignis.dfs.id in ignis.conf)
export IGNIS_DFS_ID=$(pwd)
# set docker as scheduler (ignis.scheduler.type in ignis.conf)
export IGNIS_SCHEDULER_TYPE=docker
# set where docker is available (ignis.scheduler.url in ignis.conf)
export IGNIS_SCHEDULER_URL=/var/run/docker.sock
```

The above example could be launched as follows:

```
$ ./ignis/bin/ignis-submit ignishpc/python ./driver.py
```

2 API

2.1 BasicType Reference

class Boolean
References True of False value condition.

class Integer
References a non-decimal number.

class Float
References a decimal number.

class String
References a text type.

class Json
References a json type, each language implements it in a different way.

class Pair(K, V)
References a combination of key-value types stored together as a pair.

Parameters

- **K** – Key type.
- **V** – Value type.

`class List(T)`

References a ordered collection.

Parameters **T** – Element type.

`class Map(K, V)`

References a mapping between a key and a value.

Parameters

- **K** – Key type.
- **V** – Value type.

`class Iterable(T)`

References a collection capable of returning its members one at a time.

Parameters **T** – Element type.

2.2 Driver

Ignis

The class *Ignis* manages the driver environment. Any driver function called before *Ignis.start()* and after *Ignis.stop()* will fail.

`class Ignis`

`static start()`

Starts the driver environment. The backend module is launched as a sub-process and the other driver functions can now be called. The function will not return until the entire backend configuration process has been completed.

`static stop()`

Stops the driver environment. The Backend releases all resources and finishes its execution. The function will not return until backend has finished.

IProperties

The class *IProperties* represents a persistent set of properties. Properties can be read, modified or deleted, initially instances do not contain any properties. If a property that is not stored is read, its default value will be returned if it exists.

`class IProperties`

`set(key, value)`

Sets a new property with the specified key.

Parameters

- **key** (String) – Property key.
- **value** (String) – Property value.

Returns previous value for `key` or an empty string.

Return type *String*

`get`(*key*)
Searches for the property with the specified key. If the key is not found, default value is returned.

Parameters **`key`** (`String`) – Property key.

Returns value for `key` or an empty string if it has no default value..

Return type *String*

`rm`(*key*)
Removes a property with the specified key and returns its current value.

Parameters **`key`** (`String`) – Property key.

Returns value for `key` or an empty string.

Return type *String*

`contains`(*key*)
Returns True if property with the specified key has a value or a default value.

Parameters **`key`** (`String`) – Property key.

Returns property with `key` is defined.

Return type *Boolean*

`toMap`(*defaults*)
Gets all properties and their values.

Parameters **`defaults`** (`Boolean`) – if true, unstored properties with default values are also returned.

Returns all properties and their values.

Return type *Map*(*String*, *String*)

`fromMap`(*map*)
Sets all properties defined in the argument.

Parameters **`map`** (`Map(String, String)`) – A set of properties with their values.

`load`(*path*)
Sets all properties defined in the file references by the path. The file must be formatted as .properties format where each line stores a property as `key=value` or `key:value` format.

Parameters **`path`** (`String`) – File path.

Raises ***`IDriverException`*** – An error is generated if the file does not exist, cannot be read or has an incorrect format.

`store`(*path*)
Stores all properties defined in the file references by the path.

Parameters **`path`** (`String`) – File path.

Raises ***`IDriverException`*** – An error is generated if the file cannot be created.

`clear`()
Removes all properties.

ICluster

The class *ICluster* represents a group of executors containers. Containers are identical instances with the same assigned resources, which are obtained from the properties defined in *IProperties*.

`class ICluster`(*properties*, *name*)

Parameters

- **`properties`** (`IProperties`) – Set of properties that will be used to configure the execution environment. Future modifications to the properties will have no effect.
- **`name`** (`String`) – (Optional) Gives a name to the *ICluster*, it will be used to identify the *ICluster* in the job logs and also in the Scheduler, if it supports it.

`start`()

By default, the cluster will only be started when the first computation is to be performed. This function allows you to force their creation and eliminate the time associated with requesting and granting resources. It must be used to perform performance measurements on the platform.

`destroy`()

Destroys the current running environment and frees all resources associated with it. Future executions will have to recreate the environment from scratch.

`setName`(*name*)

Sets or changes the name associated with the *ICluster*. The new name will only affect the *ICluster* log itself and future tasks created. The Scheduler and the existing tasks will keep the name used during their creation.

Parameters **`name`** (`String`) – New name.

`execute`(*args*)

Runs a command on all containers associated with the *ICluster*. This function does not trigger the creation of the *ICluster*, it will only be executed if the environment has already been created previously, otherwise the function will be registered to be invoked immediately after its creation.

Parameters **`args`** (`List(String)`) – Command and its arguments.

`executeScript`(*script*)

Like *`ICluster.execute()`* but argument is a shell script instead of single command.

Parameters **`script`** (`String`) – Linux Shell script.

`sendFile`(*source*, *target*)

Sends a file to all containers associated with the *ICluster*. This function does not trigger the creation of the *ICluster*, the file only be sent if the environment has already been created previously, otherwise the function will be registered to be invoked immediately after its creation.

Parameters

- **`source`** (`String`) – Source path in driver container.
- **`target`** (`String`) – Target path in each executor container.

`sendCompressedFile`(*source*, *target*)

Like *`ICluster.sendFile()`* but file is extracted once it has been sent. Supported formats are: `.tar`, `.tar.bz2`, `.tar.bz`, `.tar.xz`, `.tbz2`, `.tgz`, `.gz`, `.bz2`, `.xz`, `.zip`, `.Z`. Note that `.rar` is also supported, but its license requires it to be installed by the user.

ISource

The class `ISource` is an auxiliary class used by meta-functions in the driver. A meta-function is a function that defines part of its implementation using another function that is passed as a parameter. The way in which the function is defined depends on each implementation.

Typically the following format should be available:

1. *Ignis path*: String representation consisting of a file path and a class. The file indicates where the code is stored and the class defines the function to be executed. Format is as follows: `path:class`
2. *Name*: Defines only the name of the function, it is also defined as a string and differs from the previous case because it does not contain `:` separator.
3. *Source Code*: Function is defined using the syntax of the executor's source code. Executor will recognize it as source code and compile it if necessary.
4. *Lambda*: The function is defined in the driver code and then sent as bytes to the executor. In this case driver and executor must be programmed in the same programming language and it must support serialization of executable code.

`class ISource`(*function*, *native*)

Parameters

- **`function`** – Overloaded argument to accept all possible function definitions supported in each implementation.
- **`native`** (`Boolean`) – (Optional) Type of serialization used to send parameters. If true, the driver language's own serialization will be used, if and only if the executor also has the same language. Otherwise the multi-language serialization will always be used.

`addParam`(*name*, *value*)

Defines a parameter associated with the function. The value of the parameter can be obtained by the get function during its execution.

Parameters

- **`name`** (`String`) – Parameter name.
- **`value`** – Value to be stored in the parameter, can have any type.

Returns This ISource instance.

Return type *ISource*

IWorker

The class `IWorker` represents a group of processes of the same programming language. There is at least one process in each of the `ICluster` containers where the worker is created, and all containers have the same number of executor processes.

`class IWorker`(*cluster*, *type*, *name*, *cores*, *instances*)

Parameters

- **`cluster`** (`ICluster`) – `ICluster` where the executors will be created.
- **`type`** (`String`) – Name of the worker to be used, the names of the workers are associated to the programming language they execute. The available workers are associated with the image used to create the class `ICluster`.

- **`name`** (`String`) – (Optional) Like *`ICluster`* a worker can have a name that identifies it in the job log.
- **`cores`** (`Integer`) – (Optional) Number of cores associated to each executor, by default each executor uses all available cores inside the container.
- **`instances`** (`Integer`) – (Optional) Number of executors to be launched in each container, by default only one is launched.

`start`()

By default, the worker will only be started when the first computation is to be performed. This function allows you to force their creation.

`destroy`()

Destroys all processes associated with the worker. Future executions will have to start the processes again. Destroying the executors means deleting cached elements in memory, only disk cache will be kept.

`getCluster`()

gets *`ICluster`* where worker is created.

`setName`(*name*)

Sets or changes the name associated with the *`IWorker`*. The new name will only affect the worker log itself and future tasks created. Existing tasks will keep the name used during their creation.

Parameters **`name`** (`String`) – New name.

`parallelize`(*data*, *partitions*, *src*, *native*)

Creates a *`IDataFrame`* from an existing collection present in the driver. The elements present in the collection are distributed to the executors for a parallel processing.

Parameters

- **`data`** (`Iterable(T)`) – A collection object present in the driver.
- **`partitions`** (`Integer`) – How many partitions the collection elements will be divided. For optimal processing, there should be at least one partition for all cores on each of the executors.
- **`src`** (`ISource`) – (Optional) Auxiliary function to configure executor, its use may vary between languages. Must implement at least *`IBeforeFunction`* interface.
- **`native`** (`Boolean`) – (Optional) Type of serialization used to send data. If true, the driver language's own serialization will be used, if and only if the executor also has the same language. Otherwise the multi-language serialization will always be used.

Returns A parallel collection with the same type of `data` elements.

Return type *IDataFrame*(*T*)

`importDataFrame`(*data*, *src*)

Imports a parallel collection from another worker. The number of partitions will be the same as in the original worker.

Parameters

- **`data`** (`IDataFrame(T)`) – Parallel collection of source data.
- **`src`** (`ISource`) – (Optional) Auxiliary function to configure executor, its use may vary between languages. Must implement at least *`IBeforeFunction`* interface.

Returns A parallel collection with `data` elements.

Return type *IDataFrame*(*T*)

`textFile`(*path*, *minPartitions*)
Creates a parallel collection by splitting a text file to create at least `minPartitions` partitions.

Parameters

- **`path`** (`String`) – File path.
- **`minPartitions`** (`Integer`) – Minimal number of partitions.

Returns A parallel collection of strings.

Return type *IDataFrame*(*String*)

Raises ***`IDriverException`*** – An error is generated if the file does not exist or cannot be read.

`plainFile`(*path*, *minPartitions*, *delim*)
Creates a parallel collection by splitting a file using a custom delimiter to create at least `minPartitions` partitions.

Parameters

- **`path`** (`String`) – File path.
- **`minPartitions`** (`Integer`) – Minimal number of partitions. :delim String delim: A one-character string.

Returns A parallel collection of strings.

Return type *IDataFrame*(*String*)

Raises ***`IDriverException`*** – An error is generated if the file does not exist or cannot be read.

`partitionObjectFile`(*path*, *src*)
Creates a parallel collection from binary partition files. See *`IDataFrame.saveAsObjectFile()`*

Parameters

- **`path`** (`String`) – File path without the `.part*` extension.
- **`src`** (`ISource`) – (Optional) Auxiliary function to configure executor, its use may vary between languages. Must implement at east *`IBeforeFunction`* interface.

Returns A parallel collection with type stored in the binary file.

Return type *IDataFrame*(*T*)

Raises ***`IDriverException`*** – An error is generated if any file do not exist or cannot be read.

`partitionTextFile`(*path*)
Creates a parallel collection from text partition files. See *`IDataFrame.saveAsTextFile()`*

Parameters **`path`** (`String`) – File path without the `.part*` extension.

Returns A parallel collection of strings.

Return type *IDataFrame*(*String*)

Raises ***`IDriverException`*** – An error is generated if any file do not exist or cannot be read.

`partitionJsonFile`(*path*, *src*, *objectMapping*)
Creates a parallel collection from json partition files. See `IDataFrame.saveAsJsontFile()`

Parameters

- **`path`** (`String`) – File path without the `.part*` extension.
- **`src`** (`ISource`) – (Optional) Auxiliary function to configure executor, its use may vary between languages. Must implement at least *`IBeforeFunction`* interface.

- **objectMapping** (Boolean) – (Optional) If true, json objects are transformed to objects.

Returns A parallel collection of mapped object, if `objectMapping` is true or otherwise a generic json type is used.

Return type *IDataFrame*(*Json*) or *IDataFrame*(*T*)

Raises ***IDriverException*** – An error is generated if any file do not exist or cannot be read.

loadLibrary(*path*)

Loads a library of functions in the executor processes. Functions may be invoked using only their name in any *ISource*. Library type depends on the programming language of executor.

The library can be defined in two ways:

1. Path to a library file. Library must be compiled if the language requires it.
2. Source code in plain text, executor will take care of compiling if necessary. This allows you to create functions dynamically from the driver.

Parameters **path** (String) – Library path or Source code.

Raises ***IDriverException*** – An error is generated if libreary does not exist or cannot be read.

execute(*src*)

Runs a function in the executors.

Parameters **src** (*IIVoidFunction0 or* ISource) – Function to be executed.

executeTo(*src*)

Runs a function in the executors and generates a parallel collection.

Parameters **src** (IFunction0 *or* ISource) – Function to be executed.

Returns A parallel collection created with the elements returned by the function.

Return type *IDataFrame*(*T*)

call(*src*, *data*)

Runs a function that has been previously loaded by *IWorker.loadLibrary()*. Values returned by the function will generate a parallel collection. Note, this function is designed to execute functions in format *name*, it does not allow to use the other formats.

Parameters

- **src** (IFunction *or* IFunction0 *or* ISource) – Function name and its arguments. It must implement *IFunction* interface if `data` is supplied or *IFunction0* otherwise.
- **data** (IDataFrame*(*T*))* – (Optional) A parallel collection of elements to be processed by the `src` function.

Returns A parallel collection created with the elements returned by `src` function.

Return type *IDataFrame*(*T*)

voidCall(*src*, *data*)

Runs a function that has been previously loaded by *IWorker.loadLibrary()*. Like *IWorker.call()* but with no return.

Parameters

- **src** (IVoidFunction *or* IVoidFunction0 *or* ISource) – Function name and its arguments. It must implement *IVoidFunction* interface if `data` is supplied or *IVoidFunction0* otherwise. Note, this function is designed to execute functions in format *name*, it does not allow to use the other formats.

- **data** (IDataFrame(T)) – (Optional) A parallel collection of elements to be processed by the src function.

IDataFrame

The class *IDataFrame* represents a parallel collection of elements distributed among the worker executors. All functions defined within this class process the elements in a parallel and distributed way.

class IDataFrame(*T*)

class T
Represents the type associated with the parallel collection. Dynamic languages do not have to make it visible to the user, it is the input value type for most of the functions defined in *IDataFrame*.

setName(*name*)
Sets or changes the name associated with the *IDataFrame*. The new name will affect only this *IDataFrame* and future tasks created from it.

Parameters **name** (String) – New name.

persist(*cacheLevel*)
Sets a cache level for elements so that it only needs to be computed once.

Parameters **cacheLevel** (ICacheLevel) – level of cache.

cache(*cacheLevel*)
Sets a cache level *ICacheLevel.PRESERVE* for elements so that it only needs to be computed once.

unpersist()
Elements cache is disabled. Alias for *IDataFrame.uncahe*.

uncahe()
Elements cache is disabled. Alias for *IDataFrame.unpersist*.

partitions()
Gets the number of partitions.

Returns Number of partitions.

Return type Integer.

saveAsObjectFile(*path*, *compression*)
Saves elements as binary files.

Parameters

- **path** (String) – path to store the data.
- **compression** (Integer) – compresion level (0-9).

Raises ***IDriverException*** – An error is generated if files exists or cannot be write.

saveAsTextFile(*path*)
Saves elements as text files.

Parameters **path** (String) – path to store the data.

Raises ***IDriverException*** – An error is generated if files exists or cannot be write.

saveAsJsonFile(*path*, *pretty*)
Saves elements as json files.

Parameters

- **path** (String) – path to store the data.
- **pretty** (Boolean) – uses an ident format instead of compact.

Raises ***IDriverException*** – An error is generated if files exists or cannot be write.

repartition(*numPartitions*, *preserveOrdering*, *global*)
Creates a new Dataframe with a fixes number of partitions.

Parameters

- **numPartitions** (Integer) – number of partitions.
- **preserveOrdering** (Boolean) – The order of the elements does not change.
- **global** (Boolean) – Elements are balanced between different executors. If false, Elements are only balanced within each executor.

Returns A Dataframe with numPartitions.

Return type *IDataFrame*(*T*)

partitionByRandom(*numPartitions*, *seed*)
Creates a new Dataframe with a fixes number of partitions. Elements are randomly distributed among the executors.

Parameters **numPartitions** (Integer) – number of partitions. :param Integer seed: Initializes the random number generator.

Returns A Dataframe with numPartitions.

Return type *IDataFrame*(*T*)

partitionByHash(*numPartitions*)
Creates a new Dataframe with a fixes number of partitions. Elements are distributed using a hash function among the executors.

Parameters **numPartitions** (Integer) – number of partitions.

Returns A Dataframe with numPartitions.

Return type *IDataFrame*(*T*)

partitionBy(*src*, *numPartitions*)
Creates a new Dataframe with a fixes number of partitions. Elements are distributed using a custom function among the executors. The same function return assigns the same partition.

Parameters

- **src** (IFunction(T, Integer) *or ISource.*) – Function argument.
- **numPartitions** (Integer) – number of partitions.

Returns A Dataframe with numPartitions.

Return type *IDataFrame*(*T*)

map(*src*)
Performs a map operation.

Parameters **src** (IFunction(T, *R) or ISource.*) – Function argument.

Returns A Dataframe with result elements.

Return type *IDataFrame*(R)

`mapWithIndex`(*src*)

Performs a map operation. Like *`IDataFrame.map`* but global index of the element is available as the first argument of the function.

Parameters **`src`** (`IFunction2(Integer, T, R) or ISource.`) – Function argument.

Returns A Dataframe with result elements.

Return type *IDataFrame*(R)

`filter`(*src*)

Performs a filter operation. Only items that return True will be retained.

Parameters **`src`** (`IFunction(T, Boolean) or ISource.`) – Function argument.

Returns A Dataframe with result elements.

Return type *IDataFrame*(*T*)

`flatmap`(*src*)

Performs a flatmap operation. Like *`IDataFrame.map`* but each element can generate any number of results.

Parameters **`src`** (`IFunction(T, Iterable(R)) or ISource.`) – Function argument.

Returns A Dataframe with result elements.

Return type *IDataFrame*(R)

`keyBy`(*src*)

Assigns each element a key with the return of the function.

Parameters **`src`** (`IFunction(T, R) or ISource.`) – Function argument.

Returns A Dataframe of pairs with result elements.

Return type *IPairDataFrame*(R, *T*)

`mapPartitions`(*src*, *preservesPartitioning*)

Performs a specialized map that is called only once for each partition, elements can be accessed using an iterator.

Parameters

- **`src`** (`IFunction(IReadIterator(T), Iterable(R)) or ISource.`) – Function argument.
- **`preservesPartitioning`** (`Boolean`) – Preserves partitioning

Returns A Dataframe with result elements.

Return type *IDataFrame*(R)

`mapPartitionsWithIndex`(*src*, *preservesPartitioning*)

Performs a specialized map that is called only once for each partition, elements can be accessed using an iterator. Like *`IDataFrame.mapPartitions`* but global index of the partition is available as the first argument of the function.

Parameters

- **`src`** (`IFunction2(Integer, IReadIterator(T), Iterable(R)) or ISource.`) – Function argument.
- **`preservesPartitioning`** (`Boolean`) – Preserves partitioning

Returns A Dataframe with result elements.

Return type *IDataFrame*(R)

mapExecutor(*src*)

Performs a specialized map that is called only once for each executor, elements can be accessed using a list of lists where first list represents each partition. Function argument can be modified to add or remove values, if you want to generate other value type use :class: `IDataFrame.mapExecutorTo`.

Parameters **src** (`IVoidFunction(List(List(T))) or ISource.`) – Function argument.

Returns A Dataframe with result elements.

Return type *IDataFrame*(R)

mapExecutorTo(*src*)

Performs a specialized map that is called only once for each executor, elements can be accessed using a list of lists where first list represents each partition. A new list of lists must be returned to generate new partitions.

Parameters **src** (`IFunction(List(List(T)), List(List(R))) or ISource.`) – Function argument.

Returns A Dataframe with result elements.

Return type *IDataFrame*(R)

groupBy(*src*, *numPartitions*)

Groups elements that share the same key, which is obtained from the return of the function.

Parameters

- **src** (`IFunction(T, R)) or ISource.`) – Function argument.
- **numPartitions** (`Integer`) – (Optional) Number of resulting partitions.

Returns A Dataframe of pairs with result elements.

Return type *IPairDataFrame*(R, *List*(*T*))

sort(*ascending*, *numPartitions*)

Sort the elements using their natural order.

Parameters

- **ascending** (`Boolean`) – Allows you to choose between ascending and descending order.
- **numPartitions** (`Integer`) – (Optional) Number of resulting partitions.

Returns A Dataframe with result elements.

Return type *IDataFrame*(*T*)

sortBy(*src*, *ascending*, *numPartitions*)

Sort the elements using a custom function, that checks if the first argument is less than the second.

Parameters

- **src** (`IFunction2(T, T, Boolean)) or ISource.`) – Function argument.
- **ascending** (`Boolean`) – Allows you to choose between ascending and descending order.
- **numPartitions** (`Integer`) – (Optional) Number of resulting partitions.

Returns A Dataframe with result elements.

Return type *IDataFrame*(*T*)

union(*other*, *preserveOrder*, *src*)

Merges elements of two dataframes.

Parameters

- **other** (`IDataFrame(T)`) – other dataframe.
- **preserveOrder** (`Boolean`) – If true, the second dataframe is concatenated to the first, otherwise they are mixed.
- **src** (`ISource`) – (Optional) Auxiliary function to configure executor, its use may vary between languages. Must implement at east *`IBeforeFunction`* interface.

Returns A Dataframe with result elements of the two dataframes.

Return type *IDataFrame*(*T*)

`distinct`(*numPartitions*, *src*)

Duplicate elements are eliminated.

Parameters

- **numPartitions** (`Integer`) – Number of resulting partitions.
- **src** (`ISource`) – (Optional) Auxiliary function to configure executor, its use may vary between languages. Must implement at east *`IBeforeFunction`* interface.

Returns A Dataframe with result elements.

Return type *IDataFrame*(*T*)

`reduce`(*src*)

Accumulate the elements using a custom function, which must be associative and commutative. Like *`IDataFrame.treeReduce`* but final accumulation is performed in a single executor.

Parameters **src** (`IFunction2(T, T, T)) or ISource.`) – Function argument.

Returns Element resulting from accumulation.

Return type *T*

`treeReduce`(*src*)

Accumulate the elements using a custom function, which must be associative and commutative. Like *`IDataFrame.reduce`* but final accumulation is performed in parallel using multiple executors.

Parameters **src** (`IFunction2(T, T, T)) or ISource.`) – Function argument.

Returns Element resulting from accumulation.

Return type *T*

`collect`()

Retrieve all the elements.

Returns All the elements.

Return type *List*(*T*)

`aggregate`(*zero*, *seqOp*, *combOp*)

Accumulate the elements using two functions, which must be associative and commutative. Like :class: IDataFrame.treeAggregate` but final accumulation is performed in a single executor.

Parameters

- **zero** (`IFunction0(R)) or ISource.`) – Function argument to generate initial value of target type.
- **seqOp** (`IFunction2(T, R, R)) or ISource.`) – Function argument to accumulate the elements of each partition.
- **combOp** (`IFunction2(R, R, R)) or ISource.`) – Function argument to accumulate the results of all partitions .

Returns Element resulting from accumulation.

Return type R

`treeAggregate`(*zero*, *seqOp*, *combOp*)

Accumulate the elements using two functions, which must be associative and commutative. Like *`IDataFrame.aggregate`* but final accumulation is performed in parallel using multiple executors.

Parameters

- **zero** (`IFunction0` *`(R)) or ISource.`*) – Function argument to generate initial value of target type.
- **seqOp** (`IFunction2` *`(`*`T`*`, R, R)) or ISource.`*) – Function argument to accumulate the elements of each partition.
- **combOp** (`IFunction2` *`(R, R, R)) or ISource.`*) – Function argument to accumulate the results of all partitions .

Returns Element resulting from accumulation.

Return type R

`fold`(*zero*, *src*)

Accumulate the elements using a initial value and custom function, which must be associative and commutative. Like *`IDataFrame.treeFold`* but final accumulation is performed in a single executor.

Parameters

- **zero** (`IFunction0` *`(R)) or ISource.`*) – Function argument to generate initial value of target type.
- **src** (`IFunction2` *`(`*`T`*`,`* `T`*`,`* `T`*`)) or ISource.`*) – Function argument to accumulate.

Returns Element resulting from accumulation.

Return type *T*

`treeFold`(*zero*, *src*)

Accumulate the elements using a initial value and custom function, which must be associative and commutative. Like *`IDataFrame.treeFold`* but final accumulation is performed in parallel using multiple executors.

Parameters

- **zero** (`IFunction0` *`(R)) or ISource.`*) – Function argument to generate initial value of target type.
- **src** (`IFunction2` *`(`*`T`*`,`* `T`*`,`* `T`*`)) or ISource.`*) – Function argument to accumulate.

Returns Element resulting from accumulation.

Return type *T*

`take`(*num*)

Retrieves the first `num` elements.

Parameters **num** (`Integer`) – Number of elements.

Returns First `num` elements.

Return type *List*(*T*)

`foreach`(*src*)

Calls a custom function once for each element.

Parameters **src** (`IVoidFunction` *`(`*`T`*`) or ISource.`*) – Function argument.

`foreachPartition`(*src*)
Calls a custom function once for each partition, elements can be accessed using an iterator.

Parameters **src** (`IVoidFunction(IReadIterator(T)) or ISource.`) – Function argument.

`foreachExecutor`(*src*)
Calls a custom function once for each executor, elements can be accessed using a list of lists where first list represents each partition.

Parameters **src** (`IVoidFunction(List(List(T))) or ISource.`) – Function argument.

`top`(*num*, *cmp*)
Retrieves the first `num` elements in descending order. A custom function can be used to checks if the first argument is less than the second

Parameters

- **num** (`Integer`) – Number of elements.
- **cmp** (`IFunction2(T, T, Boolean)) or ISource.`) – (Optional) Comparator to be used instead of the natural order.

Returns First `num` elements.

Return type *List*(*T*)

`takeOrdered`(*num*, *cmp*)
Retrieves the first `num` elements in ascending order. A custom function can be used to checks if the first argument is less than the second

Parameters

- **num** (`Integer`) – Number of elements.
- **cmp** (`IFunction2(T, T, Boolean)) or ISource.`) – (Optional) Comparator to be used instead of the natural order.

Returns First `num` elements.

Return type *List*(*T*)

`sample`(*withReplacement*, *fraction*, *seed*)
Generates a random sample records from the original elements.

Parameters

- **withReplacement** (`Boolean`) – An element can be selected more than once.
- **fraction** (`Float`) – Percentage of the sample.
- **seed** (`Integer`) – Initializes the random number generator.

Returns A Dataframe with result elements.

Return type *IDataFrame*(*T*)

`takeSample`(*withReplacement*, *num*, *seed*)
Generates and Retrieves a random sample of `num` records from the original elements.

Parameters

- **withReplacement** (`Boolean`) – An element can be selected more than once.
- **num** (`Integer`) – Number of elements.
- **seed** (`Integer`) – Initializes the random number generator.

Returns A Dataframe with result elements.

Return type *IDataFrame*(*T*)

`count()`

Count the elements.

Returns Number of elements.

Return type *Integer*

`max(`*cmp*`)`

Retrieves the element with the maximum value. A custom function can be used to checks if the first argument is less than the second. Like `Dataframe.top` with `num=1`

Parameters

- **`num`** (`Integer`) – Number of elements.
- **`cmp`** (`IFunction2(T, T, Boolean)) or ISource.`) – (Optional) Comparator to be used instead of the natural order.

Returns Element with the maximum value.

Return type *T*

`min(`*cmp*`)`

Retrieves the element with the minimal value. A custom function can be used to checks if the first argument is less than the second. Like `Dataframe.takeOrdered` with `num=1`

Parameters

- **`num`** (`Integer`) – Number of elements.
- **`cmp`** (`IFunction2(T, T, Boolean)) or ISource .`) – (Optional) Comparator to be used instead of the natural order.

Returns Element with the minimal value.

Return type *T*

`toPair()`

Converts *`IDataFrame`* to *`IPairDataFrame`* when *`IDataFrame.T`* is a *`Pair`* of *`IPairDataFrame.K`* and *`IPairDataFrame.V`*.

Returns A Dataframe of pairs

Return type *IPairDataFrame*(*K*, *V*)

`class IPairDataFrame(`*K*, *V*`)`

Extends *`IDataFrame`* funtionality when *`IDataFrame.T`* is a *`Pair`*

`class K`

Represents the value type associated with the parallel collection. Dynamic languages do not have to make it visible to the user, it is the key input value type for most of the functions defined in *`IPairDataFrame`*.

`class V`

Represents the value type associated with the parallel collection. Dynamic languages do not have to make it visible to the user, it is the value input value type for most of the functions defined in *`IPairDataFrame`*.

`join(`*other*, *preserveOrder*, *numPartitions*, *src*`)`

Joins an element of this collection with an element of `other` that share the same key.

Parameters

- **`other`** (`IPairDataFrame(K, V)`) – other dataframe.

- **numPartitions** (Integer) – Number of resulting partitions.
- **src** (ISource) – (Optional) Auxiliary function to configure executor, its use may vary between languages. Must implement at east *IBeforeFunction* interface.

Returns A Dataframe of pairs with result elements.

Return type *IPairDataFrame*(*K*, *Pair*(*V*, *V*))

flatMapValues(*src*)

Performs a map function only on the values while preserving the key. Like *IPairDataFrame.mapValues* but each element can generate any number of results, key is duplicated or deleted if necessary.

Parameters **src** (IFunction*(*V*, R) or ISource.*) – Function argument.

Returns A Dataframe of pairs with result elements.

Return type *IPairDataFrame*(*K*, R)

mapValues(*src*)

Performs a map function only on the values while preserving the key.

Parameters **src** (IFunction*(*V*, R) or ISource.*) – Function argument.

Returns A Dataframe of pairs with result elements.

Return type *IPairDataFrame*(*K*, R)

groupByKey(*numPartitions*, *src*)

Groups elements that share the same key.

Parameters

- **numPartitions** (Integer) – Number of resulting partitions.
- **src** (ISource) – (Optional) Auxiliary function to configure executor, its use may vary between languages. Must implement at east *IBeforeFunction* interface.

Returns A Dataframe of pairs with result elements.

Return type *IPairDataFrame*(*K*, *List*(*V*))

reduceByKey(*src*, *numPartitions*, *localReduce*)

Accumulate the values that share the same key using a custom function, which must be associative and commutative.

Parameters

- **src** (IFunction2*(*V, V, V*)) or ISource.*) – Function argument.
- **numPartitions** (Integer) – Number of resulting partitions.
- **localReduce** (Boolean) – Accumulate the values that share the same key in a executor before global accumulation. Reduces the size of the exchange if there are duplicated keys in multiple partitions.

Returns A Dataframe of pairs with result elements.

Return type *IPairDataFrame*(*K*, *V*)

aggregateByKey(*zero*, *seqOp*, *combOp*, *numPartitions*)

Accumulate the values that share the same key using two functions, which must be associative and commutative.

Parameters

- **zero** (`IFunction0(R)) or ISource.`) – Function argument to generate initial value of target type.
- **seqOp** (`IFunction2(V, R, R)) or ISource.`) – Function argument to accumulate the values that share the same key of each partition.
- **combOp** (`IFunction2(R, R, R)) or ISource.`) – Function argument to accumulate the results that share the same key of all partitions .
- **numPartitions** (`Integer`) – Number of resulting partitions.

Returns A Dataframe of pairs with result elements.

Return type *IPairDataFrame*(*K*, *V*)

`foldByKey`(*zero*, *src*, *numPartitions*, *localFold*)

Accumulate the values that share the same key using a initial value and custom function, which must be associative and commutative.

Parameters

- **zero** (`IFunction0(R)) or ISource.`) – Function argument to generate initial value of target type.
- **src** (`IFunction2(V, V, V)) or ISource.`) – Function argument to accumulate.
- **numPartitions** (`Integer`) – Number of resulting partitions.
- **localFold** (`Boolean`) – Accumulate the values that share the same key in a executor before global accumulation. Reduces the size of the exchange if there are duplicated keys in multiple partitions.

Returns A Dataframe of pairs with result elements.

Return type *IPairDataFrame*(*K*, *V*)

`sortByKey`(*ascending*, *numPartitions*, *src*)

Sort the keys using their natural order.

Parameters

- **ascending** (`Boolean`) – Allows you to choose between ascending and descending order.
- **numPartitions** (`Integer`) – Number of resulting partitions.
- **src** (`ISource`) – (Optional) Auxiliary function to configure executor, its use may vary between languages. Must implement at east *`IBeforeFunction`* interface.

Returns A Dataframe of pairs with result elements.

Return type *IPairDataFrame*(*K*, *V*)

`keys()`

Retrieve unique keys.

Returns The unique keys.

Return type *List*(*K*)

`values()`

Retrieve unique values.

Returns The unique values.

Return type *List*(*V*)

`sampleByKey`(*withReplacement*, *fractions*, *seed*, *native*)

Generates a random sample records from the values that share the same key.

Parameters

- **`withReplacement`** (`Boolean`) – An element can be selected more than once.
- **`fraction`** (`Map(K, Float)`) – Percentage of the sample by key. Absences are taken as zero.
- **`seed`** (`Integer`) – Initializes the random number generator.
- **`native`** (`Boolean`) – (Optional) sends `fractions` with native serialization.

Returns A Dataframe with result elements.

Return type *IDataFrame*(*T*)

`countByKey`()

Count unique keys.

Returns Number unique of values.

Return type *Integer*

`countByValue`()

Count unique keys.

Returns Number unique of values.

Return type *Integer*

`class ICacheLevel`

`NO_CACHE: Integer = 0`

Elements cache is disabled.

`PRESERVE: Integer = 1`

Elements will be cached in the same storage in which it is stored.

`MEMORY: Integer = 2`

Elements will be cached on memory storage.

`RAW_MEMORY: Integer = 3`

Elements will be cached on raw memory storage.

`DISK: Integer = 4`

Elements will be cached on disk storage.

IDriverException

The class `IDriverException` represents an execution error. Exceptions are defined together with the function that generates them, but they are actually thrown by the function that causes the execution.

`class IDriverException`

2.3 Executor

class IContext

The executor context allows the API functions to interact with the rest of the IgnisHPC system.

cores()

Returns Number of cores assigned to the executor.

Return type *Integer*

executors()

Returns Number of executors.

Return type *Integer*

executorId()

Returns Unique identifier of the executor, a number greater than or equal to zero and less than the number of executors.

Return type *Integer*

threadId()

Returns Unique identifier of the current thread, a number greater than or equal to zero and less than the than the number of cores.

Return type *Integer*

mpiGroup()

Returns Returns the mpi group of the executors.

props()

Returns Driver `IProperties` as `Map` object.

Return type *Map*(*String*, *String*)

vars()

(This function may vary depending on the implementation.)

Returns Variables sent by `ISource.addParam` as `Map` object.

Return type *Map*(*String*, Any)

class IReadIterator

Transverse through elements of a partition.

hasNext()

Returns True if elements remain

Return type *Boolean*

next()

Returns Next element.

class IBeforeFunction

`before`(*context*)

Parameters **context** (`IContext`) – The executor context.

`class IVoidFunction0`

`before`(*context*)

Parameters **context** (`IContext`) – The executor context.

`call`(*context*)

Parameters **context** (`IContext`) – The executor context.

`after`(*context*)

Parameters **context** (`IContext`) – The executor context.

`class IVoidFunction`

`before`(*context*)

Parameters **context** (`IContext`) – The executor context.

`call`(*context*, *v*)

Parameters

- **context** (`IContext`) – The executor context.
- **v** – Argument

`after`(*context*)

Parameters **context** (`IContext`) – The executor context.

`class IVoidFunction2`

`before`(*context*)

Parameters **context** (`IContext`) – The executor context.

`call`(*context*, *v1*, *v2*)

Parameters

- **context** (`IContext`) – The executor context.
- **v1** – Argument 1
- **v2** – Argument 2

`after`(*context*)

Parameters **`context`** (`IContext`) – The executor context.

`class IFunction0`

`before`(*context*)

Parameters **`context`** (`IContext`) – The executor context.

`call`(*context*)

Parameters **`context`** (`IContext`) – The executor context.

Returns This function must return a value.

`after`(*context*)

Parameters **`context`** (`IContext`) – The executor context.

`class IFunction`

`before`(*context*)

Parameters **`context`** (`IContext`) – The executor context.

`call`(*context*, *v*)

Parameters

- **`context`** (`IContext`) – The executor context.
- **`v`** – Argument

Returns This function must return a value.

`after`(*context*)

Parameters **`context`** (`IContext`) – The executor context.

`class IFunction2`

`before`(*context*)

Parameters **`context`** (`IContext`) – The executor context.

`call`(*context*, *v1*, *v2*)

Parameters

- **`context`** (`IContext`) – The executor context.
- **`v1`** – Argument 1
- **`v2`** – Argument 2

Returns This function must return a value.

after(*context*)

Parameters **context** (IContext) – The executor context.

3 Docker Images

3.1 Background

IgnisHPC is fully containerized, these can be extended to create custom runtime environments and isolate incompatible modules. For those unfamiliar with containers, containers are running instances of an image, which is an immutable environment that serves as the starting point when the container is started.

IgnisHPC images are organized according to a hierarchy, shown in the figure above, which allows us to extend and create new modules easily and simply. The images have an associated namespace that groups all the hierarchy of images, by default it is **ignishpc** but an user can create his own. Although all images have an important role to play, in most cases, users should only familiarize themselves with the core images. A core container stores the IgnisHPC implementation for a given language as well as its dependencies. Images without an associated IgnisHPC module are stored in the Dockerfiles repository.

Note that only images belonging to the IgnisHPC architecture are covered, images of external dependencies are optional and are outside the scope of this document.

3.2 Details

Base

Base, as its name indicates, is the base image of IgnisHPC. Base is created as an extension of the official ubuntu image whose version is associated with the IgnisHPC version. This image only defines environment variables and creates the IgnisHPC folder structure, which can be customized and modify the other images easily.

Builder

Builder extends from base and installs the most common dependencies used in compilation environments such as the GNU Compiler Collection, the GNU Debugger and other libraries and development tools needed for software compilation. This image and its derivatives are only used for software construction, they are never used for code execution.

common-Builder

In this image the dependencies common to the whole system are stored and compiled. MPI is compiled and modified from sources in this image and the thrift library is stored so that the cores do not have to download it.

driver-Builder

This image is associated with the driver environment build, which compiles the source code and defines an installation script to install it in the driver runtime environment. This image belongs to the Backend module and both are stored in the same repository.

executor-Builder

This image is associated with the executor environment build, there is no module associated to the executor, so only the installation script is created to install it in the executor runtime environment and compiles any dependencies if necessary.

common

Common execution environment for the execution of all IgnisHPC modules, this image extends from base to inherit the environment configuration. The image build depends on driver-builder and exectutor-builder, from which it obtains the dependencies and installation scripts for a driver and executor environment. Neither the dependencies nor the runtime environments are installed, they are just stored.

core-builder

The programming languages supported by IgnisHPC are known as cores, which consist of a driver code and an executor code. Each core has a builder image used for the compilation of sources and its dependencies and for the creation of an installation script. The name of the image must be the name of the core with the suffix `-builder` and must be stored in the same repository as the core source code.

core images

Core images are the execution environments used by users to run their codes in IgnisHPC. The core images are three and are generated automatically from their corresponding builder extending the common image.

1. It has the same name as the core with suffix `-driver`, it has the driver environment installed.
2. It has the same name as the core with suffix `-executor`, it has the executor environment installed.
3. It has the same name as the core, it has the driver and executor environment installed.

The core images have the core and its dependencies installed but can only be used as a driver or executor if its environment is installed.

core helper images

In some cases it is possible that a core may need an additional image. For example, compiled languages may define a `-compiler` image to facilitate the compilation process. These images are stored in the source code repository and should be prefixed with the core name to identify them.

full

A full runtime Image, like the cores, is generated automatically. This image has the enviroment driver, the enviroment executor and all available cores installed. This is the default image for the executors when no image is selected.

submitter

This image is associated with the IgnisHPC job launch, with are launched using the `ignis-submit` script. The submiter module is implemented together with the backend module, so both are compiled in the `driver-builder` and the image is stored in the backend repository.

4 Properties

IgnisHPC properties control most of the application settings and are configured separately for each application. Properties can be defined dynamically in the driver code, in the submitter script or as an environment variable. Default values are stored in both cases in `/opt/ignis/etc/ignis.conf`.

All the system variables start with the prefix *ignis*. They can be of different types:

- Read-Only variables: contain information about the current job (`ignis.home`, `ignis.job.name`, `ignis.job.directory`, ...)
- Driver/submitter variables: must be defined before launching the driver, any later modification will have no effect on the system (`ignis.driver.image`, `ignis.driver.port`, `ignis.driver.cores`, ...)
- Executors variables: can be defined by all the available methods. Once the execution environment is created, modifications will have no effect (`ignis.executor.cores`, `ignis.partition.type`, `ignis.modules.io.compression`, ...)

4.1 How to set properties?

Driver code

Driver has a Properties object available that allows users to read and write properties. The default values can be overwritten but will recover their value if the property is deleted.

See the driver section for more details.

Enviroment variable

Driver and submitter scan the environment variables for properties at startup. Environment variables starting with `IGNIS_` will be treated as properties. The variable names will be converted to lowercase and the `_` will be converted to `.`. For example, `IGNIS_FOO_BAR` will be stored as `ignis.foo.bar`, whose value will remain unchanged.

Submitter script

The submitter sets the properties values using the `-p` or `--properties` parameter when a job is launched.

See submitter section for more details.

File

Default values are stored in `/opt/ignis/etc/ignis.conf` using Java Properties with a `key=value` format.

Mixed definition

In case of multiple definitions of the same variable, the following priority list will be used:

1. Driver Properties Object (highest priority)
2. Driver Container enviroment variable
3. Submitter script argument
4. Submitter Container enviroment variable
5. File `/opt/ignis/etc/ignis.conf`*

* There are two different files in the driver and submitter, and each one only stores the default values for its module. Note that IgnisHPC does not define any default value outside the configuration files, which allows users to know the values of all the system variables. Therefore, values can be modified but never deleted.

4.2 Property list

Base Properties

Name	Type	Default	Context	Description
ignis.debug	Boolean	False	All	Enables debugging messages.
ignis.home	Path	`auto`	Driver Executor	Path where IgnisHPC is installed, IgnisHPC images set this value to `/opt/ignis`.
ignis.options	Raw	`auto`	Driver	Raw value used by the submitter to send options to the driver.
ignis.working.directory	Path	`auto`	Driver Executor	Working directory of the current Job, it is used to resolve all relative paths.

Job Properties

Name	Type	Default	Context	Description
ignis.job.id	String	`auto`	Driver	Job identifier, generated and used by the scheduler.
ignis.job.name	String	`auto`	Driver	Job name, can be sent as a submitter parameter, otherwise it will be generated automatically. The final job name may vary depending on the scheduler used.
ignis.job.directory	Path	`auto`	Executor	Directory where the job data is stored, it is generated as a subdirectory of the working directory.
ignis.job.worker	Integer	`auto`	Executor	Identifies the worker whose instance is the executor.

Distributed Filesystem (DFS) Properties

Name	Type	Default	Context	Description
ignis.dfs.id	String	`auto`	Driver	DFS identifier, it identifies DFS in the scheduler.
ignis.dfs.home	Path	`auto`	Driver Executor	Directory where the DFS is mounted on.

Scheduler Properties

Name	Type	Default	Context	Description
ignis.scheduler.url	URL[]	`auto`	Driver	One or more Scheduler API URL, syntax is Scheduler-dependent.
ignis.scheduler.type	String	`auto`	Driver	Scheduler implementation name. See Scheduler section for value names.
ignis.scheduler.dns	String[]	`auto`	Driver	Hostnames to resolve in the container network.
ignis.scheduler.param.{name}	String		Driver	It sets <name> parameter for the Scheduler, each Scheduler has its own parameters.

Driver Properties

Name	Type	Default	Context	Description
ignis.driver.image	String	`empty`	Driver	Driver: container image
ignis.driver.cores	Interger	1	Driver	Driver: number of cores
ignis.driver.memory	String	1GB	Driver	Driver: memory limit in Bytes, might use prefixes (K, M, G, …) or (Ki, Mi, Gi, …).
ignis.driver.rpc.port	Port	4000	Driver	Backend service listening port.
ignis.driver.rpc.compression	Integer	6	Driver	Backend service RPC zlib compression level. (0-9)
ignis.driver.swappiness	Integer	`empty`	Driver	Driver: Container swappiness rate. (0-100)
ignis.driver.pool	Integer	8	Driver	Minimum number of workers on standby when the Backend is idle.
ignis.driver.port.{tcp|udp}.{cport}	Port		Driver	Driver: exposes a container port to a host port. Value `0` generates a random host port.
ignis.driver.ports.{tcp|udp}	Integer		Driver	Driver: exposes a specific number of random ports to the host, ports are exposed to the same value on host .
ignis.driver.bind.{cpath}	Path		Driver	Driver: binds a container path `cpath` to a host path. Add ':ro' for read-only.
ignis.driver.volume.{cpath}	String		Driver	Driver: Creates a volume in the path with value size in Bytes, might use prefixes (K, M, G, …) or (Ki, Mi, Gi, …).
ignis.driver.hosts	String[]	`empty`	Driver	Driver: the container must be launched on one of the hosts in order of preference.
ignis.driver.env.{name}	String	`empty`	Driver	Driver: creates an environment variable in the container.
ignis.driver.public.key	String	`auto`	Driver	SSH tunnel public key.
ignis.driver.private.key	String	`auto`	Driver Executor	SSH tunnel private key.
ignis.driver.healthcheck.port	String	1963	Driver	Backend healthcheck listening port.
ignis.driver.healthcheck.url	String	`auto`	Driver Executor	Backend healthcheck URL.
ignis.driver.healthcheck.interval	Integer	60	Driver Executor	How often the driver is checked to see if it is still alive.
ignis.driver.healthcheck.timeout	Integer	20	Driver Executor	Backend healthcheck response timeout.
ignis.driver.healthcheck.retries	Integer	5	Driver Executor	Number of healthcheck failures before aborting.

Executor Properties

Name	Type	Default	Context	Description
ignis.executor.instances	Integer	1	Executor	Number of executors.
ignis.executor.attempts	Integer	2	Executor	Number of execution attempts before failure.
ignis.executor.image	String	ignishpc /full	Executor	Executor: container image.
ignis.executor.cores	Interger	1	Executor	Executor: number of cores.
ignis.executor.cores.single	String[]	python	Executor	Executors that do not support multithreading. Threads are transformed into processes.
ignis.executor.memory	String	1GB	Executor	Executor: memory limit in Bytes, might use prefixes (K, M, G, …) or (Ki, Mi, Gi, …).
ignis.executor.rpc.port	Port	5000	Executor	Executor service listening port.
ignis.executor.rpc.compression	Integer	6	Executor	Executor service RPC zlib compression level. (0-9)
ignis.executor.swappiness	Integer	0	Executor	Executor: container swappiness rate. (0-100)
ignis.executor.isolation	Boolean	True	Executor	Prevents different workers from running in the same container at the same time.
ignis.executor.directory	Path	`auto`	Executor	Directory where the job data is stored, it is generated as a subdirectory of job directory.
ignis.executor.port.{tcp|udp} .{cport}	Port		Executor	Executor: exposes a container port to a host port. Value `0` generates a random host port.
ignis.executor.ports. {tcp|udp}	Integer		Executor	Executor: exposes a specific number of random ports to the host, ports are exposed to the same value on host.
ignis.executor.bind.{cpath}	Path		Executor	Executor: binds a container path `cpath` to a host path. Add ':ro' to value for read-only.
ignis.executor.volume.{cpath}	String		Executor	Executor: creates a volume in the path with value size in Bytes, might use prefixes (K, M, G, …) or (Ki, Mi, Gi, …).
ignis.executor.hosts	String[]	`empty`	Executor	Executor: the container must be launched on one of the hosts in order of preference.
ignis.executor.env.{name}	String	`empty`	Executor	Executor: creates an environment variable in the container.

Partition Properties

Name	Type	Default	Context	Description
ignis.partition.type	String	Memory	Executor	Storage type for partitions, must be `Memory`, `RawMemory` or `Disk`.
ignis.partition.minimal	String	128MB	Executor	Minimum partition size from file.
ignis.partition.compression	Integer	0	Executor	Storage zlib compresion level. Available for `RawMemory` and `Disk`. (0-9)
ignis.partition.serialization	String	native	Executor	Type of serialization with executors of the same language.

Transport Properties

Name	Type	Default	Context	Description
ignis.transport.cores	Float	0.0	Executor	Number of threads used to execute a transport action at the same time. If the value is less than 1, the value will be multiplied by `ignis.executor.cores`.
ignis.transport.compression	Integer	0	Executor	Transport zlib compresion level. (0-9)
ignis.transport.ports	Integer	20	Executor	Number of ports reserved for data exchanges.
ignis.transport.minimal	String	100KB	Executor	Minimum size to open a data transport channel, otherwise it will be sent by RPC.
ignis.transport.element.size	String	256B	Executor	Average size per element to use as a reference when it cannot be calculated.

Module Properties

Name	Type	Default	Context	Description
ignis.modules.io.compression	Integer	0	Executor	File zlib compresion level. (0-9)
ignis.modules.io.cores	Float	0.0	Executor	Number of threads used to read/write files at the same time. If the value is less than 1, the value will be multiplied by `ignis.executor.cores`.
ignis.transport.compression	Integer	0	Executor	Transport zlib compresion level. (0-9)
ignis.modules.io.overwrite	Boolean	False	Executor	Output files are overwritten if they already exist.
ignis.modules.sort.samples	Float	0.001	Executor	Sampling size in the sort algorithm. Number of samples is calculated using this value and the number of elements. If the value is greater than 1, it will be used as the number of samples.
ignis.modules.sort.resampling	Boolean	False	Executor	Samples from the sort algorithm are resampled for parallel processing. It is only useful if large amounts of data are sorted or if the sample size is very high.
ignis.modules.exchange.type	String	auto	Executor	Algorithm used for data exchange, can be sync or async. Any other value selects the method that best fits.

Index