pyDive Documentation

Release 1.1 beta

Heiko Burau

June 22, 2015

CONTENTS

Contents:

CHAPTER

ONE

GETTING STARTED

1.1 Quickstart

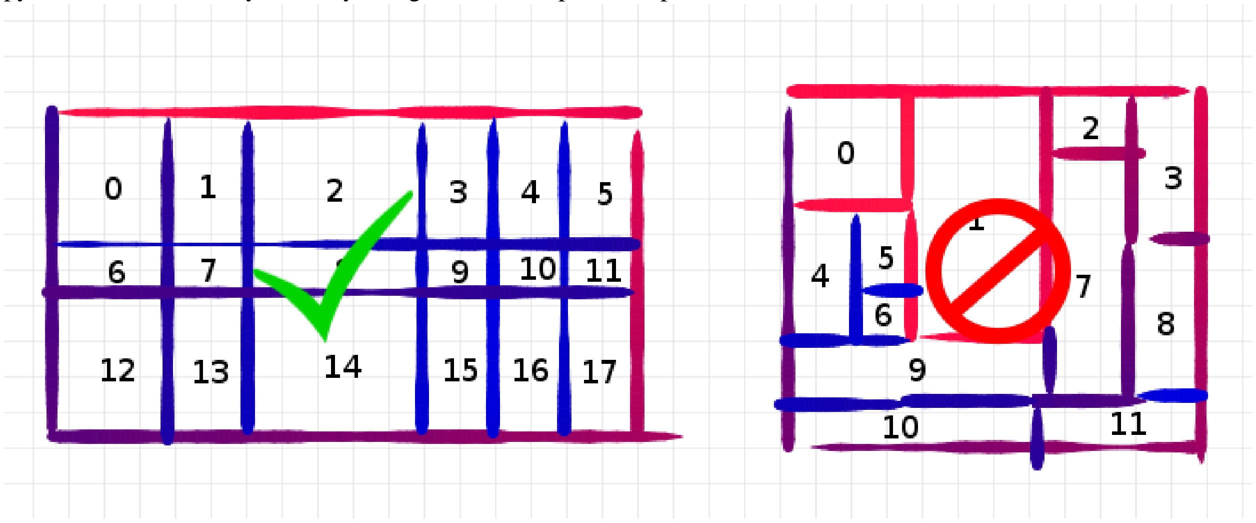
pyDive is built on top of *IPython.parallel*, *numpy* and *mpi4py*. *h5py*, *adios* and *pycuda* are optional. Running `python setup.py install` will install pyDive with these and other required packages from *requirements.txt*. Alternatively you can install it via pip: `pip install pyDive`.

Basic code example:

```
import pyDive
pyDive.init(profile='mpi')

arrayA = pyDive.ones((1000, 1000, 1000), distaxes='all')
arrayB = pyDive.zeros_like(arrayA)

# do some array operations, + - * / sin cos, ..., slicing, etc...
arrayC = arrayA + arrayB

# plot result
import matplotlib.pyplot as plt
plt.imshow(arrayC[500,::10,::10])
```

Before actually running this script there must have been an IPython.parallel cluster launched (see section below) otherwise *pyDive.init()* fails.

pyDive distributes array-memory along one or multiple user-specified axes:



You can either specify the exact decomposition for each axis or leave the default which persuits to squared chunks.

Although the array elements are stored on the cluster nodes you have full access through indexing. If you want to have a local array from a pyDive-array anyway you can call `array.gather()` but make sure that your pyDive-array is small enough to fit into your local machine's memory. If not you may want to slice it first. Note that an array is also gathered implicitly if you try to access an attribute which is only available for the local array. This is why there is no `gather()` in the example above when calling `imshow()`.

1.2 Setup an IPython.parallel cluster configuration

The first step is to create an IPython.parallel profile: http://ipython.org/ipython-doc/2/parallel/parallel_process.html. The name of this profile is the argument of `pyDive.init()`. It defaults to `"mpi"`. Starting the cluster is then the second and final step:

```
$ ipcluster start -n 4 --profile=mpi
```

1.3 Run tests

In order to test the pyDive installation run:

```
$ python setup.py test
```

This will ask you for the IPython.parallel profile to be used and the number of engines to be started, e.g.:

```
$ Name of your IPython-parallel profile you want to run the tests with: pbs
$ Number of engines: 256
```

Then the script starts the cluster, runs the tests and finally stops the cluster. If you have already a cluster running by your own you can also run the tests by launching `py.test` from the pyDive directory and setting the environment variable `IPP_PROFILE_NAME` to the profile's name.

1.4 Overview

pyDive knows different kinds of distributed arrays, all corresponding to a local, non-distributed array:

- numpy -> *`pyDive.ndarray`* -> Stores array elements in cluster nodes' memory.
- hdf5 -> `pyDive.h5.h5_ndarray` -> Stores array elements in a hdf5-file.
- adios -> `pyDive.ad.ad_ndarray` -> Stores array elements in a adios-file.
- gpu -> `pyDive.gpu.gpu_ndarray` -> Stores array elements in clusters' gpus.
- `pyDive.cloned_ndarray` -> Holds independent copies of one array on cluster nodes.

Among these three packages there are a few modules:

- `pyDive.structered` -> structured datatypes
- *`pyDive.algorithm`* -> map, reduce, mapReduce
- *`pyDive.fragment`* -> fragment file-disk array to fit into the cluster's main memory
- *`pyDive.mappings`* -> particle-mesh mappings
- *`pyDive.picongpu`* -> helper functions for picongpu-users
- *`pyDive.pyDive`* -> shortcuts for most used functions

CHAPTER

TWO

TUTORIALS

In this section we are going through a few use cases for pyDive. If you want to test the code you can download the `sample hdf5-file`. It has the following dataset structure:

```
$ h5ls -r sample.h5
/                        Group
/fields                  Group
/fields/fieldB           Group
/fields/fieldB/z         Dataset {256, 256}
/fields/fieldE           Group
/fields/fieldE/x         Dataset {256, 256}
/fields/fieldE/y         Dataset {256, 256}
/particles               Group
/particles/cellidx       Group
/particles/cellidx/x     Dataset {10000}
/particles/cellidx/y     Dataset {10000}
/particles/pos           Group
/particles/pos/x         Dataset {10000}
/particles/pos/y         Dataset {10000}
/particles/vel           Group
/particles/vel/x         Dataset {10000}
/particles/vel/y         Dataset {10000}
/particles/vel/z         Dataset {10000}
```

After launching the cluster (*Setup an IPython.parallel cluster configuration*) the first step is to initialize pyDive:

```
import pyDive
pyDive.init()
```

Load a single dataset:

```
h5fieldB_z = pyDive.h5.open("sample.h5", "/fields/fieldB/z", distaxes='all')

assert type(h5fieldB_z) is pyDive.h5.h5_ndarray
```

h5fieldB_z just holds a dataset *handle*. To read out data into memory call `load()`:

```
fieldB_z = h5fieldB_z.load()

assert type(fieldB_z) is pyDive.ndarray
```

This loads the entire dataset into the main memory of all *engines*. The array elements are distributed along all axes.

We can also load a hdf5-group:

```
h5fieldE = pyDive.h5.open("sample.h5", "/fields/fieldE", distaxes='all')
fieldE = h5fieldE.load()
```

h5fieldE and *fieldE* are some so called "virtual array-of-structures", see: `pyDive.structered`.

```
>>> print h5fieldE
VirtualArrayOfStructs<array-type: <class 'pyDive.distribution.multiple_axes.h5_ndarray'>, shape: [25
  y -> float32
  x -> float32

>>> print fieldE
VirtualArrayOfStructs<array-type: <class 'pyDive.distribution.multiple_axes.ndarray'>, shape: [256, 2
  y -> float32
  x -> float32
```

Now, let's do some calculations!

2.1 Example 1: Total field energy

Computing the total field energy of an electromagnetic field means squaring and summing or in pyDive's words:

```
import pyDive
import numpy as np
pyDive.init()

h5input = "sample.h5"

h5fields = pyDive.h5.open(h5input, "/fields") # defaults to distaxes='all'
fields = h5fields.load() # read out all fields into cluster's main memory in parallel

energy_field = fields.fieldE.x**2 + fields.fieldE.y**2 + fields.fieldB.z**2

total_energy = pyDive.reduce(energy_field, np.add)
print total_energy
```

Output:

```
$ python example1.py
557502.0
```

Well this was just a very small hdf5-sample of 1.3 MB however in real world we deal with a lot greater data volumes. So what happens if *h5fields* is too large to be stored in the main memory of the whole cluster? The line `fields = h5fields.load()` will crash. In this case we want to load the hdf5 data piece by piece. The function *pyDive.fragment* helps us doing so:

```
import pyDive
import numpy as np
pyDive.init()

h5input = "sample.h5"

big_h5fields = pyDive.h5.open(h5input, "/fields")
# big_h5fields.load() # would cause a crash

total_energy = 0.0
for h5fields in pyDive.fragment(big_h5fields):
    fields = h5fields.load()

    energy_field = fields.fieldE.x**2 + fields.fieldE.y**2 + fields.fieldB.z**2
    total_energy += pyDive.reduce(energy_field, np.add)

print total_energy
```

An equivalent way to get this result is a `pyDive.mapReduce`:

```
...
def square_fields(h5fields):
    fields = h5fields.load()
    return fields.fieldE.x**2 + fields.fieldE.y**2 + fields.fieldB.z**2

total_energy = pyDive.mapReduce(square_fields, np.add, h5fields)
print total_energy
```

square_fields is called on each *engine* where *h5fields* is a structure (*pyDive.arrayOfStructs*) of `h5_ndarrays` representing a sub part of the big *h5fields*. *pyDive.algorithm.mapReduce()* can be called with an arbitrary number of arrays including `pyDive.ndarrays`, `pyDive.h5.h5_ndarrays`, `pyDive.adios.ad_ndarrays` and `pyDive.cloned_ndarrays`. If there are `pyDive.h5.h5_ndarrays` or `pyDive.adios.ad_ndarrays` it will check whether they fit into the combined main memory of all cluster nodes as a whole and loads them piece by piece if not.

Now let's say our dataset is really big and we just want to get a first estimate of the total energy:

```
...
total_energy = pyDive.mapReduce(square_fields, np.add, h5fields[::10, ::10]) * 10.0**2
```

Slicing on pyDive-arrays is always allowed.

If you use picongpu here is an example of how to get the total field energy for each timestep (see *pyDive.picongpu*):

```
import pyDive
import numpy as np
pyDive.init()

def square_field(h5field):
    field = h5field.load()
    return field.x**2 + field.x**2 + field.x**2

for step, h5field in pyDive.picongpu.loadAllSteps("/.../simOutput", "fields/FieldE"):
    total_energy = pyDive.mapReduce(square_field, np.add, h5field)

    print step, total_energy
```

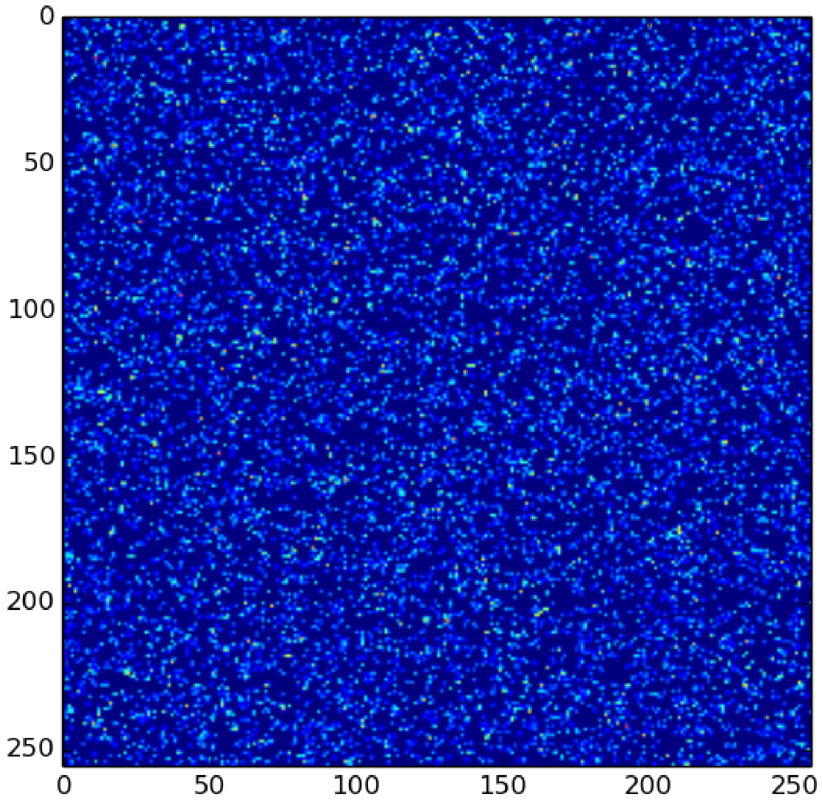
2.2 Example 2: Particle density field

Given the list of particles in our `sample.h5` we want to create a 2D density field out of it. For this particle-to-mesh mapping we need to apply a certain particle shape like cloud-in-cell (CIC), triangular-shaped-cloud (TSC), and so on. A list of these together with the actual mapping functions can be found in the *pyDive.mappings* module. If you miss a shape you can easily create one by your own by defining a particle shape function. Note that if you have numba installed the shape function will be compiled resulting in a significant speed-up.

We assume that the particle positions are distributed randomly. This means although each engine is loading a separate part of all particles it needs to write to the entire density field. Therefore the density field must have a whole representation on each participating engine. This is the job of *pyDive.cloned_ndarray.cloned_ndarray.cloned_ndarray*.

```
import pyDive
import numpy as np
pyDive.init()

shape = [256, 256]
density = pyDive.cloned.zeros(shape)

h5input = "sample.h5"

particles = pyDive.h5.open(h5input, "/particles")

def particles2density(particles, density):
    particles = particles.load()
    total_pos = particles.cellidx.astype(np.float32) + particles.pos

    # convert total_pos to an (N, 2) shaped array
    total_pos = np.hstack((total_pos.x[:,np.newaxis],
                           total_pos.y[:,np.newaxis]))

    par_weighting = np.ones(particles.shape)
    import pyDive.mappings
    pyDive.mappings.particles2mesh(density, par_weighting, total_pos, pyDive.mappings.CIC)

pyDive.map(particles2density, particles, density)

final_density = density.sum() # add up all local copies

from matplotlib import pyplot as plt
plt.imshow(final_density)
plt.show()
```

Output:



Here, as in the first example, *particles2density* is a function executed on the *engines* by `pyDive.algorithm.map()`. All of its arguments are numpy-arrays or structures (`pyDive.arrayOfStructs`) of numpy-arrays.

`pyDive.algorithm.map()` can also be used as a decorator:

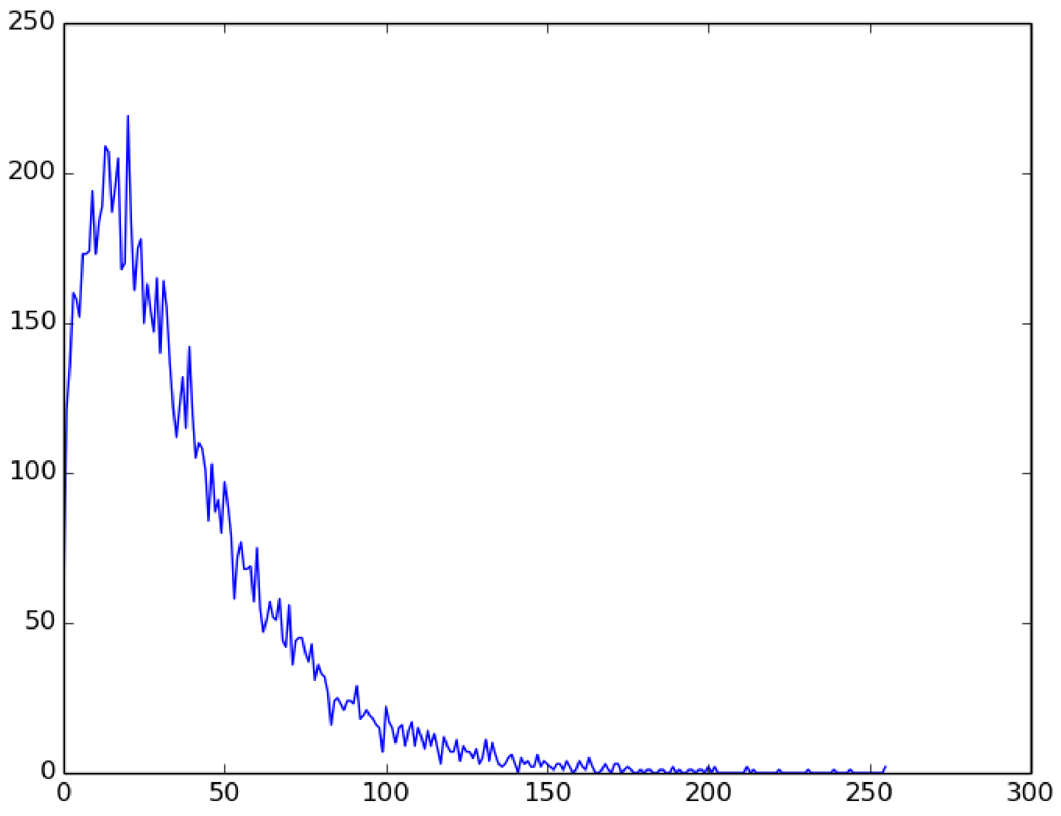
```
@pyDive.map
def particles2density(particles, density):
    ...

particles2density(particles, density)
```

2.3 Example 3: Particle energy spectrum

```
import pyDive
import numpy as np
pyDive.init()

bins = 256
spectrum = pyDive.cloned.zeros([bins])

h5input = "sample.h5"

velocities = pyDive.h5.open(h5input, "/particles/vel")
```

```
@pyDive.map
def vel2spectrum(velocities, spectrum, bins):
    velocities = velocities.load()
    mass = 1.0
    energies = 0.5 * mass * (velocities.x**2 + velocities.y**2 + velocities.z**2)

    spectrum[:], bin_edges = np.histogram(energies, bins)

vel2spectrum(velocities, spectrum, bins=bins)

final_spectrum = spectrum.sum() # add up all local copies

from matplotlib import pyplot as plt
plt.plot(final_spectrum)
plt.show()
```

Output:



CHAPTER

THREE

REFERENCE

3.1 Arrays

3.1.1 pyDive.arrays.ndarray module

Note: All of this module's functions and classes are also directly accessable from the `pyDive` module.

pyDive.ndarray class

class `pyDive.`**`ndarray`**(*shape*, *dtype=<type 'float'>*, *distaxes='all'*, *target_offsets=None*, *target_ranks=None*, *no_allocation=False*, ***kwargs*)

Represents a cluster-wide, multidimensional, homogeneous array of fixed-size elements. *cluster-wide* means that its elements are distributed across IPython.parallel-engines. The distribution is done in one or multiply dimensions along user-specified axes. The user can optionally specify which engine maps to which index range or leave the default that persuits an uniform distribution across all engines.

This **ndarray** - class is auto-generated out of its local counterpart: **numpy.ndarray**.

The implementation is based on IPython.parallel and local numpy.ndarray - arrays. Every special operation numpy.ndarray implements ("__add__", "__le__", ...) is also available for ndarray.

Note that array slicing is a cheap operation since no memory is copied. However this can easily lead to the situation where you end up with two arrays of the same size but of distinct element distribution. Therefore call dist_like() first before doing any manual stuff on their local arrays. However every cluster-wide array operation first equalizes the distribution of all involved arrays, so an explicit call to dist_like() is rather unlikely in most use cases.

If you try to access an attribute that is only available for the local array, the request is forwarded to an internal local copy of the whole distributed array (see: `gather()`). This internal copy is only created when you want to access it and is held until `__setitem__` is called, i.e. the array's content is manipulated.

`__init__`(*shape*, *dtype=<type 'float'>*, *distaxes='all'*, *target_offsets=None*, *target_ranks=None*, *no_allocation=False*, ***kwargs*)

Creates an instance of ndarray. This is a low-level method of instantiating an array, it should rather be constructed using factory functions ("empty", "zeros", "open", ...)

Parameters

- **`shape`** (*ints*) – shape of array
- **`dtype`** – datatype of a single element
- **`distaxes`** (*ints*) – distributed axes. Accepts a single integer too. Defaults to 'all' meaning each axis is distributed.

- **`target_offsets`** (*list of lists*) – For each distributed axis there is a (inner) list in the outer list. The inner list contains the offsets of the local array.
- **`target_ranks`** (*ints*) – linear list of *engine* ranks holding the local arrays. The last distributed axis is iterated over first.
- **`no_allocation`** (*bool*) – if `True` no instance of numpy.ndarray will be created on engine. Useful for manual instantiation of the local array.
- **`kwargs`** – additional keyword arguments are forwarded to the constructor of the local array.

`copy`()

Returns a hard copy of this array.

`dist_like`(*other*)

Redistributes a copy of this array (*self*) like *other* and returns the result. Checks whether redistribution is necessary and returns *self* if not.

Redistribution involves inter-engine communication.

Parameters **`other`** (*distributed array*) – target array

Raises AssertionError if the shapes of *self* and *other* don't match.

Returns new array with the same content as *self* but distributed like *other*. If *self* is already distributed like *other* nothing is done and *self* is returned.

`gather`()

Gathers local instances of numpy.ndarray from *engines*, concatenates them and returns the result.

Note: You may not call this method explicitly because if you try to access an attribute of the local array (numpy.ndarray), `gather()` is called implicitly before the request is forwarded to that internal gathered array. Just access attributes like you do for the local array. The internal copy is held until `__setitem__` is called, e.g. `a[1] = 3.0`, setting a dirty flag to the local copy.

Warning: If another array overlapping this array is manipulating its data there is no chance to set the dirty flag so you have to keep in mind to call `gather()` explicitly in this case!

Returns instance of numpy.ndarray

Factory functions

These are convenient functions to create a *pyDive.ndarray* instance.

`pyDive.arrays.ndarray.`**`array`**(*array_like*, *distaxes='all'*)

Create a pyDive.ndarray instance from an array-like object.

Parameters

- **`array_like`** – Any object exposing the array interface, e.g. numpy-array, python sequence, ...
- **`distaxes`** (*ints*) – distributed axes. Defaults to 'all' meaning each axis is distributed.

`pyDive.arrays.ndarray.`**`empty`**(*shape*, *dtype=<type 'float'>*, *distaxes='all'*, ***kwargs*)

Create a *ndarray* instance. This function calls its local counterpart *numpy.empty* on each *engine*.

Parameters

- **`shape`** (*ints*) – shape of array

- **`dtype`** – datatype of a single element
- **`distaxes`** (*ints*) – distributed axes
- **`kwargs`** – keyword arguments are passed to the local function *numpy.empty*

`pyDive.arrays.ndarray.`**`empty_like`**`(`*other*, ***kwargs*`)`

Create a *ndarray* instance with the same shape, dtype and distribution as `other`. This function calls its local counterpart *numpy.empty_like* on each *engine*.

Parameters

- **`other`** – other array
- **`kwargs`** – keyword arguments are passed to the local function *numpy.empty_like*

`pyDive.arrays.ndarray.`**`hollow`**`(`*shape*, *dtype=<type 'float'>*, *distaxes='all'*`)`

Create a pyDive.ndarray instance distributed across all engines without allocating a local numpy-array.

Parameters

- **`shape`** (*ints*) – shape of array
- **`dtype`** – datatype of a single element
- **`distaxes`** (*ints*) – distributed axes. Defaults to 'all' meaning each axis is distributed.

`pyDive.arrays.ndarray.`**`hollow_like`**`(`*other*`)`

Create a pyDive.ndarray instance with the same shape, distribution and type as `other` without allocating a local numpy-array.

`pyDive.arrays.ndarray.`**`zeros`**`(`*shape*, *dtype=<type 'float'>*, *distaxes='all'*, ***kwargs*`)`

Create a *ndarray* instance. This function calls its local counterpart *numpy.zeros* on each *engine*.

Parameters

- **`shape`** (*ints*) – shape of array
- **`dtype`** – datatype of a single element
- **`distaxes`** (*ints*) – distributed axes
- **`kwargs`** – keyword arguments are passed to the local function *numpy.zeros*

`pyDive.arrays.ndarray.`**`zeros_like`**`(`*other*, ***kwargs*`)`

Create a *ndarray* instance with the same shape, dtype and distribution as `other`. This function calls its local counterpart *numpy.zeros_like* on each *engine*.

Parameters

- **`other`** – other array
- **`kwargs`** – keyword arguments are passed to the local function *numpy.zeros_like*

`pyDive.arrays.ndarray.`**`ones`**`(`*shape*, *dtype=<type 'float'>*, *distaxes='all'*, ***kwargs*`)`

Create a *ndarray* instance. This function calls its local counterpart *numpy.ones* on each *engine*.

Parameters

- **`shape`** (*ints*) – shape of array
- **`dtype`** – datatype of a single element
- **`distaxes`** (*ints*) – distributed axes
- **`kwargs`** – keyword arguments are passed to the local function *numpy.ones*

`pyDive.arrays.ndarray.`**`ones_like`**`(`*other*, ***kwargs*`)`

Create a *ndarray* instance with the same shape, dtype and distribution as `other`. This function calls its local counterpart *numpy.ones_like* on each *engine*.

Parameters

- **`other`** – other array
- **`kwargs`** – keyword arguments are passed to the local function *numpy.ones_like*

Universal functions

numpy knows the so called *ufuncs* (universal function). These are functions which can be applied elementwise on an array, like *sin*, *cos*, *exp*, *sqrt*, etc. All of these *ufuncs* from *numpy* are also available for *pyDive.ndarray* arrays, e.g.

```
a = pyDive.ones([100])
a = pyDive.sin(a)
```

3.1.2 pyDive.arrays.h5_ndarray module

Note: This module has a shortcut: `pyDive.h5`.

class `pyDive.arrays.h5_ndarray.`**`h5_ndarray`**`(`*shape*, *dtype=<type 'float'>*, *distaxes='all'*, *target_offsets=None*, *target_ranks=None*, *no_allocation=False*, ***kwargs*`)`

Represents a cluster-wide, multidimensional, homogeneous array of fixed-size elements. *cluster-wide* means that its elements are distributed across IPython.parallel-engines. The distribution is done in one or multiply dimensions along user-specified axes. The user can optionally specify which engine maps to which index range or leave the default that persuits an uniform distribution across all engines.

This **h5_ndarray** - class is auto-generated out of its local counterpart: **pyDive.arrays.local.h5_ndarray.h5_ndarray**.

The implementation is based on IPython.parallel and local pyDive.arrays.local.h5_ndarray.h5_ndarray - arrays. Every special operation pyDive.arrays.local.h5_ndarray.h5_ndarray implements ("__add__", "__le__", ...) is also available for h5_ndarray.

Note that array slicing is a cheap operation since no memory is copied. However this can easily lead to the situation where you end up with two arrays of the same size but of distinct element distribution. Therefore call dist_like() first before doing any manual stuff on their local arrays. However every cluster-wide array operation first equalizes the distribution of all involved arrays, so an explicit call to dist_like() is rather unlikely in most use cases.

If you try to access an attribute that is only available for the local array, the request is forwarded to an internal local copy of the whole distributed array (see: `gather()`). This internal copy is only created when you want to access it and is held until `__setitem__` is called, i.e. the array's content is manipulated.

`__init__``(`*shape*, *dtype=<type 'float'>*, *distaxes='all'*, *target_offsets=None*, *target_ranks=None*, *no_allocation=False*, ***kwargs*`)`

Creates an instance of h5_ndarray. This is a low-level method of instantiating an array, it should rather be constructed using factory functions ("empty", "zeros", "open", ...)

Parameters

- **`shape`** (*ints*) – shape of array
- **`dtype`** – datatype of a single element

- **distaxes** (*ints*) – distributed axes. Accepts a single integer too. Defaults to 'all' meaning each axis is distributed.
- **target_offsets** (*list of lists*) – For each distributed axis there is a (inner) list in the outer list. The inner list contains the offsets of the local array.
- **target_ranks** (*ints*) – linear list of *engine* ranks holding the local arrays. The last distributed axis is iterated over first.
- **no_allocation** (*bool*) – if `True` no instance of pyDive.arrays.local.h5_ndarray.h5_ndarray will be created on engine. Useful for manual instantiation of the local array.
- **kwargs** – additional keyword arguments are forwarded to the constructor of the local array.

load()
Load array from file into main memory of all engines in parallel.

Returns pyDive.ndarray instance

pyDive.arrays.h5_ndarray.**open**(*filename*, *datapath*, *distaxes='all'*)
Create an pyDive.h5.h5_ndarray instance respectively a structure of pyDive.h5.h5_ndarray instances from file.

Parameters

- **filename** – name of hdf5 file.
- **dataset_path** – path within hdf5 file to a single dataset or hdf5 group.
- **ints** (*distaxes*) – distributed axes. Defaults to 'all' meaning each axis is distributed.

Returns pyDive.h5.h5_ndarray instance / structure of pyDive.h5.h5_ndarray instances

pyDive.arrays.h5_ndarray.**open_dset**(*filename*, *dataset_path*, *distaxes='all'*)
Create a pyDive.h5.h5_ndarray instance from file.

Parameters

- **filename** – name of hdf5 file.
- **dataset_path** – path within hdf5 file to a single dataset.
- **ints** (*distaxes*) – distributed axes. Defaults to 'all' meaning each axis is distributed.

Returns pyDive.h5.h5_ndarray instance

3.1.3 pyDive.arrays.ad_ndarray module

Note: This module has a shortcut: `pyDive.adios`.

3.1.4 pyDive.arrays.gpu_ndarray module

Note: This module has a shortcut: `pyDive.gpu`.

class pyDive.arrays.gpu_ndarray.**gpu_ndarray**(*shape*, *dtype=<type 'float'>*, *distaxes='all'*, *target_offsets=None*, *target_ranks=None*, *no_allocation=False*, ***kwargs*)
Represents a cluster-wide, multidimensional, homogeneous array of fixed-size elements. *cluster-wide* means that its elements are distributed across IPython.parallel-engines. The distribution is done in one or multiply

dimensions along user-specified axes. The user can optionally specify which engine maps to which index range or leave the default that persuits an uniform distribution across all engines.

This **gpu_ndarray** - class is auto-generated out of its local counterpart: **pyDive.arrays.local.gpu_ndarray.gpu_ndarray**.

The implementation is based on IPython.parallel and local pyDive.arrays.local.gpu_ndarray.gpu_ndarray - arrays. Every special operation pyDive.arrays.local.gpu_ndarray.gpu_ndarray implements ("__add__", "__le__", ...) is also available for gpu_ndarray.

Note that array slicing is a cheap operation since no memory is copied. However this can easily lead to the situation where you end up with two arrays of the same size but of distinct element distribution. Therefore call dist_like() first before doing any manual stuff on their local arrays. However every cluster-wide array operation first equalizes the distribution of all involved arrays, so an explicit call to dist_like() is rather unlikely in most use cases.

If you try to access an attribute that is only available for the local array, the request is forwarded to an internal local copy of the whole distributed array (see: `gather()`). This internal copy is only created when you want to access it and is held until `__setitem__` is called, i.e. the array's content is manipulated.

`__init__`(*shape*, *dtype=<type 'float'>*, *distaxes='all'*, *target_offsets=None*, *target_ranks=None*, *no_allocation=False*, ***kwargs*)

Creates an instance of gpu_ndarray. This is a low-level method of instantiating an array, it should rather be constructed using factory functions ("empty", "zeros", "open", ...)

Parameters

- **`shape`** (*ints*) – shape of array
- **`dtype`** – datatype of a single element
- **`distaxes`** (*ints*) – distributed axes. Accepts a single integer too. Defaults to 'all' meaning each axis is distributed.
- **`target_offsets`** (*list of lists*) – For each distributed axis there is a (inner) list in the outer list. The inner list contains the offsets of the local array.
- **`target_ranks`** (*ints*) – linear list of *engine* ranks holding the local arrays. The last distributed axis is iterated over first.
- **`no_allocation`** (*bool*) – if `True` no instance of pyDive.arrays.local.gpu_ndarray.gpu_ndarray will be created on engine. Useful for manual instantiation of the local array.
- **`kwargs`** – additional keyword arguments are forwarded to the constructor of the local array.

`to_cpu`()

Copy array data to cpu main memory.

Result pyDive.ndarray distributed cpu array.

`pyDive.arrays.gpu_ndarray.`**`array`**(*array_like*, *distaxes='all'*)

Create a pyDive.gpu_ndarray instance from an array-like object.

Parameters

- **`array_like`** – Any object exposing the array interface, e.g. numpy-array, python sequence, ...
- **`distaxis`** (*ints*) – distributed axes. Defaults to 'all' meaning each axis is distributed.

`pyDive.arrays.gpu_ndarray.`**`hollow`**(*shape*, *dtype=<type 'float'>*, *distaxes='all'*)

Create a pyDive.gpu_ndarray instance distributed across all engines without allocating a local gpu-array.

Parameters

- **shape** (*ints*) – shape of array
- **dtype** – datatype of a single element
- **distaxes** (*ints*) – distributed axes. Defaults to 'all' meaning each axis is distributed.

`pyDive.arrays.gpu_ndarray.`**`hollow_like`**(*other*)

Create a pyDive.gpu_ndarray instance with the same shape, distribution and type as `other` without allocating a local gpu-array.

3.1.5 pyDive.cloned_ndarray package

Submodules

pyDive.cloned_ndarray.cloned_ndarray module

class `pyDive.cloned_ndarray.cloned_ndarray.`**`cloned_ndarray`**(*shape*, *dtype=<type 'float'>*, *target_ranks='all'*, *no_allocation=False*)

Represents a multidimensional, homogenous array of fixed-size elements which is cloned on the cluster nodes. *Cloned* means that every participating *engine* holds an independent, local numpy-array of the user-defined shape. The user can then do e.g. some manual stuff on the local arrays or some computation with `pyDive.algorithm` on them.

Note that there exists no 'original' array as the name might suggest but something like that can be generated by `merge()`.

`__init__`(*shape*, *dtype=<type 'float'>*, *target_ranks='all'*, *no_allocation=False*)

Creates an `pyDive.cloned_ndarray.cloned_ndarray.cloned_ndarray` instance. This is a low-level method for instanciating a cloned_array. Cloned arrays should be constructed using 'empty', 'zeros' or 'empty_targets_like' (see `pyDive.cloned_ndarray.factories`).

Parameters

- **shape** (*ints*) – size of the array on each axis
- **dtype** (*numpy-dtype*) – datatype of a single data value
- **target_ranks** (*ints*) – list of *engine*-ids that share this array. Or 'all' for all engines.
- **no_allocation** (*bool*) – if `True` no actual memory, i.e. *numpy-array*, will be allocated on *engine*. Useful when you want to assign an existing numpy array manually.

`merge`(*op*)

Merge all local arrays in a pair-wise operation into a single numpy-array.

Parameters **op** – Merging operation. Expects two numpy-arrays and returns one.

Returns merged numpy-array.

`sum`()

Add up all local arrays.

Returns numpy-array.

pyDive.cloned_ndarray.factories module

This module holds high-level functions for instanciating pyDive.cloned_ndarrays.

`pyDive.cloned_ndarray.factories.`**`empty`**`(`*shape*, *dtype=<type 'float'>*`)`

Return a new *pyDive.cloned_ndarray package* utilizing all engines without initializing elements.

Parameters

- **`shape`** (*ints*) – shape of the array
- **`dtype`** (*numpy-dtype*) – datatype of a single data value

`pyDive.cloned_ndarray.factories.`**`empty_engines_like`**`(`*shape*, *dtype*, *a*`)`

Return a new `pyDive.cloned_ndarray` utilizing the same engines *a* does without initializing elements.

Parameters

- **`shape`** (*ints*) – shape of the array
- **`dtype`** (*numpy-dtype*) – datatype of a single data value
- **`a`** – *pyDive.arrays.ndarray module*

`pyDive.cloned_ndarray.factories.`**`hollow`**`(`*shape*, *dtype=<type 'float'>*`)`

Return a new *pyDive.cloned_ndarray package* utilizing all engines without allocating a local *numpy-array*.

Parameters

- **`shape`** (*ints*) – shape of the array
- **`dtype`** (*numpy-dtype*) – datatype of a single data value

`pyDive.cloned_ndarray.factories.`**`hollow_engines_like`**`(`*shape*, *dtype*, *a*`)`

Return a new `pyDive.cloned_ndarray` utilizing the same engines *a* does without allocating a local *numpy-array*.

Parameters

- **`shape`** (*ints*) – shape of the array
- **`dtype`** (*numpy-dtype*) – datatype of a single data value
- **`a`** – *pyDive.arrays.ndarray module*

`pyDive.cloned_ndarray.factories.`**`ones`**`(`*shape*, *dtype=<type 'float'>*`)`

Return a new *pyDive.cloned_ndarray package* utilizing all engines filled with ones.

Parameters

- **`shape`** (*ints*) – shape of the array
- **`dtype`** (*numpy-dtype*) – datatype of a single data value

`pyDive.cloned_ndarray.factories.`**`zeros`**`(`*shape*, *dtype=<type 'float'>*`)`

Return a new *pyDive.cloned_ndarray package* utilizing all engines filled with zeros.

Parameters

- **`shape`** (*ints*) – shape of the array
- **`dtype`** (*numpy-dtype*) – datatype of a single data value

`pyDive.cloned_ndarray.factories.`**`zeros_engines_like`**`(`*shape*, *dtype*, *a*`)`

Return a new *pyDive.cloned_ndarray package* utilizing the same engines *a* does filled with zeros.

Parameters

- **shape** (*ints*) – shape of the array
- **dtype** (*numpy-dtype*) – datatype of a single data value
- **a** – *pyDive.arrays.ndarray module*

3.2 Modules

Note: All functions of these modules are also directly accessable from the `pyDive` module.

3.2.1 pyDive.arrayOfStructs module

The *arrayOfStructs* module addresses the common problem when dealing with structured data: While the user likes an array-of-structures layout the machine prefers a structure-of-arrays. In pyDive the method of choice is a *virtual array-of-structures*-object. It holds array-like attributes such as shape and dtype and allows for slicing but is operating on a structure-of-arrays internally.

Example:

```
...
treeOfArrays = {"FieldE" :
                    {"x" : fielde_x,
                     "y" : fielde_y,
                     "z" : fielde_z},
                "FieldB" :
                    {"x" : fieldb_x,
                     "y" : fieldb_y,
                     "z" : fieldb_z}
                }

fields = pyDive.arrayOfStructs(treeOfArrays)

half = fields[::2]["FieldE/x"]
# equivalent to
half = fields["FieldE/x"][::2]
# equivalent to
half = fields["FieldE"]["x"][::2]
# equivalent to
half = fields["FieldE"][::2]["x"]

# equivalent to
half = fields.FieldE.x[::2]
```

The example shows that in fact *fields* can be treated as an array-of-structures **or** a structure-of-arrays depending on what is more appropriate.

The goal is to make the virtual *array-of-structs*-object look like a real array. Therefore every method call or operation is forwarded to the individual arrays.:

```
new_field = fields.FieldE.astype(np.int) + fields.FieldB.astype(np.float)
```

Here the forwarded method calls are `astype` and `__add__`.

`pyDive.arrayOfStructs.`**`arrayOfStructs`**`(`*structOfArrays*`)`
Convert a *structure-of-arrays* into a virtual *array-of-structures*.

Parameters **`structOfArrays`** – tree-like dictionary of arrays.

Raises

- **`AssertionError`** – if the *arrays-types* do not match. Datatypes may differ.
- **`AssertionError`** – if the shapes do not match.

Returns Custom object representing a virtual array whose elements have the same tree-like structure as *structOfArrays*.

3.2.2 pyDive.algorithm module

`pyDive.algorithm.`**`map`**(*f*, **arrays*, ***kwargs*)

Applies *f* on *engine* on local arrays related to *arrays*. Example:

```
cluster_array = pyDive.ones(shape=[100], distaxes=0)

cluster_array *= 2.0
# equivalent to
pyDive.map(lambda a: a *= 2.0, cluster_array) # a is the local numpy-array of *cluster_array*
```

Or, as a decorator:

```
@pyDive.map
def twice(a):
    a *= 2.0

twice(cluster_array)
```

Parameters

- **`f`** (*callable*) – function to be called on *engine*. Has to accept *numpy-arrays* and *kwargs*
- **`arrays`** – list of arrays including *pyDive.ndarrays*, *pyDive.h5_ndarrays* or *pyDive.cloned_ndarrays*
- **`kwargs`** – user-specified keyword arguments passed to *f*

Raises

- **`AssertionError`** – if the *shapes* of *pyDive.ndarrays* and *pyDive.h5_ndarrays* do not match
- **`AssertionError`** – if the *distaxes* attributes of *pyDive.ndarrays* and *pyDive.h5_ndarrays* do not match

Notes:

- If the hdf5 data exceeds the memory limit (currently 25% of the combined main memory of all cluster nodes) the data will be read block-wise so that a block fits into memory.
- *map* chooses the list of *engines* from the **first** element of *arrays*. On these engines *f* is called. If the first array is a *pyDive.h5_ndarray* all engines will be used.
- *map* is not writing data back to a *pyDive.h5_ndarray* yet.
- *map* does not equalize the element distribution of *pyDive.ndarrays* before execution.

pyDive.algorithm.**mapReduce**(*map_func*, *reduce_op*, **arrays*, ***kwargs*)

Applies *map_func* on *engine* on local arrays related to *arrays* and reduces its result in a tree-like fashion over all axes. Example:

```
cluster_array = pyDive.ones(shape=[100], distaxes=0)

s = pyDive.mapReduce(lambda a: a**2, np.add, cluster_array) # a is the local numpy-array of *clu
assert s == 100
```

Parameters

- **f** (*callable*) – function to be called on *engine*. Has to accept *numpy-arrays* and *kwargs*
- **reduce_op** (*numpy-ufunc*) – reduce operation, e.g. *numpy.add*.
- **arrays** – list of arrays including *pyDive.ndarrays*, *pyDive.h5_ndarrays* or *pyDive.cloned_ndarrays*
- **kwargs** – user-specified keyword arguments passed to *f*

Raises

- **AssertionError** – if the *shapes* of *pyDive.ndarrays* and *pyDive.h5_ndarrays* do not match
- **AssertionError** – if the *distaxes* attributes of *pyDive.ndarrays* and *pyDive.h5_ndarrays* do not match

Notes:

- If the hdf5 data exceeds the memory limit (currently 25% of the combined main memory of all cluster nodes) the data will be read block-wise so that a block fits into memory.
- *mapReduce* chooses the list of *engines* from the **first** element of *arrays*. On these engines the mapReduce will be executed. If the first array is a *pyDive.h5_ndarray* all engines will be used.
- *mapReduce* is not writing data back to a *pyDive.h5_ndarray* yet.
- *mapReduce* does not equalize the element distribution of *pyDive.ndarrays* before execution.

pyDive.algorithm.**reduce**(*array*, *op*)

Perform a tree-like reduction over all axes of *array*.

Parameters

- **array** – *pyDive.ndarray*, *pyDive.h5_ndarray* or *pyDive.cloned_ndarray* to be reduced
- **op** (*numpy-ufunc*) – reduce operation, e.g. *numpy.add*.

If the hdf5 data exceeds the memory limit (currently 25% of the combined main memory of all cluster nodes) the data will be read block-wise so that a block fits into memory.

3.2.3 pyDive.fragment module

pyDive.fragment.**fragment**(**arrays*, ***kwargs*)

Create fragments of *arrays* so that each fragment will fit into the combined main memory of all engines when calling `load()`. The fragmentation is done by array slicing along the longest axis of `arrays[0]`. The edge size of the fragments is a power of two except for the last fragment.

Parameters

- **array** – distributed arrays (e.g. pyDive.ndarray, pyDive.h5_ndarray, ...)

- **kwargs** – optional keyword arguments are: `memory_limit` and `offset`.
- **memory_limit** (*float*) – fraction of the combined main memory of all engines reserved for fragmentation. Defaults to `0.25`.
- **offset** (*bool*) – If `True` the returned tuple is extended by the fragments' offset (along the distributed axis). Defaults to `False`.

Raises

- **AssertionError** – If not all arrays have the same shape.
- **AssertionError** – If not all arrays are distributed along the same axis.

Returns generator object (list) of tuples. Each tuple consists of one fragment for each array in *arrays*.

Note that *arrays* may contain an arbitrary number of distributed arrays of any type. While the fragments' size is solely calculated based on the memory consumption of arrays that store their elements on hard disk (see *hdd_arraytypes*), the fragmentation itself is applied on all arrays in the same way.

Example:

```
big_h5_array = pyDive.h5.open("monster.h5", "/")
# big_h5_array.load() # crash

for h5_array, offset in pyDive.fragment(big_h5_array, offset=True):
    a = h5_array.load() # no crash
    print "This fragment's offset is", offset, "on axis:", a.distaxis
```

`pyDive.fragment.`**hdd_arraytypes = (<class 'pyDive.distribution.multiple_axes.h5_ndarray'>, None)**
list of array types that store their elements on hard disk

3.2.4 pyDive.mappings module

If numba is installed the particle shape functions will be compiled which gives an appreciable speedup.

class `pyDive.mappings.`**CIC**
Cloud-in-Cell

class `pyDive.mappings.`**NGP**
Nearest-Grid-Point

`pyDive.mappings.`**mesh2particles**(*mesh*, *particles_pos*, *shape_function=<class pyDive.mappings.CIC>*)
Map mesh values to particles according to a particle shape function.

Parameters

- **mesh** (*array-like*) – n-dimensional array. Dimension of *mesh* has to be greater or equal to the number of particle position components.
- **particles_pos** (*(N, d)*) – 'd'-dim tuples for 'N' particle positions. The positions can be float32 or float64 and must be within the shape of *mesh*.
- **shape_function** (*callable, optional*) – Callable object returning the particle assignment value for a given param 'x'. Has to provide a 'support' float attribute which defines the width of the non-zero area. Defaults to cloud-in-cell.

Returns Mapped mesh values for each particle.

Notes:

- The particle shape function is not evaluated outside the mesh.

`pyDive.mappings.`**`particles2mesh`**(*mesh*, *particles*, *particles_pos*, *shape_function=<class pyDive.mappings.CIC>*)

Map particle values to mesh according to a particle shape function. Particle values are added to the mesh.

Parameters

- **`mesh`** (*array-like*) – n-dimensional array. Dimension of *mesh* has to be greater or equal to the number of particle position components.
- **`particles`** (*array_like (1 dim)*) – particle data. len(*particles*) has to be the same as len(*particles_pos*)
- **`particles_pos`** (*(N, d)*) – 'd'-dim tuples for 'N' particle positions. The positions can be float32 or float64 and must be within the shape of *mesh*.
- **`shape_function`** (*callable, optional*) – Callable object returning the particle assignment value for a given param 'x'. Has to provide a 'support' float attribute which defines the width of the non-zero area. Defaults to cloud-in-cell.

Returns *mesh*

Notes:

- The particle shape function is not evaluated outside the mesh.

3.2.5 pyDive.picongpu module

This module holds convenient functions for those who use pyDive together with picongpu.

`pyDive.picongpu.`**`getSteps`**(*folder_path*)

Returns a list of all timesteps in *folder_path*.

`pyDive.picongpu.`**`loadAllSteps`**(*folder_path*, *data_path*, *distaxis=0*)

Python generator object looping hdf5-data of all timesteps found in *folder_path*.

This generator doesn't read or write any data elements from hdf5 but returns dataset-handles covered by *pyDive.h5_ndarray* objects.

All datasets within *data_path* must have the same shape.

Parameters

- **`folder_path`** (*str*) – Path to the folder containing the hdf5-files
- **`data_path`** (*str*) – Relative path starting from "/data/<timestep>/" within hdf5-file to the dataset or group of datasets
- **`distaxis`** (*int*) – axis on which datasets are distributed over when once loaded into memory.

Returns tuple of timestep and a pyDive.h5_ndarray or a structure of pyDive.h5_ndarrays (`pyDive.structured`). Ordering is done by timestep.

Notes:

- If the dataset has a '**sim_unit**' attribute its value is stored in `h5array.unit`.

`pyDive.picongpu.`**`loadStep`**(*step*, *folder_path*, *data_path*, *distaxis=0*)

Load hdf5-data from a single timestep found in *folder_path*.

All datasets within *data_path* must have the same shape.

Parameters

- **`step`** (*int*) – timestep
- **`folder_path`** (*str*) – Path to the folder containing the hdf5-files
- **`data_path`** (*str*) – Relative path starting from "/data/<timestep>/" within hdf5-file to the dataset or group of datasets
- **`distaxis`** (*int*) – axis on which datasets are distributed over when once loaded into memory.

Returns pyDive.h5_ndarray or a structure of pyDive.h5_ndarrays (`pyDive.structured`).

Notes:

- If the dataset has a '**sim_unit**' attribute its value is stored in `h5array.unit`.

`pyDive.picongpu.`**`loadSteps`**(*steps*, *folder_path*, *data_path*, *distaxis=0*)

Python generator object looping all hdf5-data found in *folder_path* from timesteps appearing in *steps*.

This generator doesn't read or write any data elements from hdf5 but returns dataset-handles covered by *pyDive.h5_ndarray* objects.

All datasets within *data_path* must have the same shape.

Parameters

- **`steps`** (*ints*) – list of timesteps to loop
- **`folder_path`** (*str*) – Path to the folder containing the hdf5-files
- **`data_path`** (*str*) – Relative path starting from "/data/<timestep>/" within hdf5-file to the dataset or group of datasets
- **`distaxis`** (*int*) – axis on which datasets are distributed over when once loaded into memory.

Returns tuple of timestep and a pyDive.h5_ndarray or a structure of pyDive.h5_ndarrays (`pyDive.structured`). Ordering is done by timestep.

Notes:

- If the dataset has a '**sim_unit**' attribute its value is stored in `h5array.unit`.

3.2.6 pyDive.pyDive module

Make most used functions and modules directly accessable from pyDive.

Functions:

`abs`

`absolute`

`add`

`arccos`

`arccosh`

arcsin

arcsinh

arctan

arctan2

arctanh

array

bitwise_and

bitwise_not

bitwise_or

bitwise_xor

ceil

conj

conjugate

copysign

cos

cosh

deg2rad

degrees

divide

empty

empty_like

equal

exp

exp2

expm1

fabs

floor

floor_divide

fmax

fmin

fmod

fragment

frexp

greater

greater_equal

hollow

hollow_like

hypot

init

invert

isfinite

isinf

isnan

ldexp

left_shift

less

less_equal

log

log10

log1p

log2

logaddexp

logaddexp2

logical_and

logical_not

logical_or

logical_xor

map

mapReduce

maximum

mesh2particles

minimum

mod

modf

multiply

ndarray

negative

nextafter

not_equal

ones

ones_like

particles2mesh

power

rad2deg

radians

reciprocal

reduce

remainder

right_shift

rint

sign

signbit

sin

sinh

spacing

sqrt

square

structured

subtract

tan

tanh

true_divide

trunc

zeros

zeros_like

Modules:

IPParallelClient

algorithm

arrays

cloned

cloned_ndarray

h5

mappings

picongpu

CHAPTER FOUR

INDICES AND TABLES

- genindex
- modindex
- search

engine The cluster nodes of *IPython.parallel* are called *engines*. Sometimes they are also called *targets*. They are the workers of pyDive performing all the computation and file i/o and they hold the actual array-memory. From the user perspective you don't to deal with them directly.