pyDive Documentation

Release

Heiko Burau

September 08, 2014

CONTENTS

Contents:

CHAPTER

ONE

GETTING STARTED

1.1 Quickstart

pyDive is built on top of *IPython.parallel*, *numpy*, *mpi4py* and *h5py*. Running `python setup.py install` will install pyDive with these and other required packages from *requirements.txt*.

Basic code example:

```
import pyDive
pyDive.init()

arrayA = pyDive.ones([1000, 1000, 1000], distaxis=0)
arrayB = pyDive.zeros_like(arrayA)

# do some array operations, + - * / sin cos, ..., slicing, etc...
...

# get numpy-array
result = arrayC.gather()
# plot result
...
```

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

To keep things simple pyDive distributes array-memory only along **one** user-specified axis. This axis is given by the *distaxis* parameter at array instanciation. It should usually be the largest axis in order to have the best surface-to-volume ratio. But keep in mind that during arithmetic operations both arrays have to be distributed along the *same* axis.

Although the array elements are stored on the cluster nodes you have full access through indexing. If you want to have a numpy-array from a pyDive-array anyway you can call the method `arrayC.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.

1.2 Setup an IPython.parallel cluster configuration

The first step is to create an IPython.parallel profile in MPI-mode: 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 Overview

pyDive knows three kinds of arrays associated to a separate python package respectively:

- `pyDive.ndarray` -> Stores array elements in cluster nodes' memory.
- `pyDive.h5_ndarray` -> Stores array elements in a hdf5-file.
- `pyDive.cloned_ndarray` -> Holds independent copies of one array on cluster nodes.

Among these three packages there are a few modules:

- `pyDive.arrayOfStructs` -> structured datatypes
- `pyDive.algorithm` -> map, reduce, mapReduce
- `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.fromPath("sample.h5", "/fields/fieldB/z", distaxis=0)

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

h5fieldB_z just holds a dataset *handle*. To read out data into memory do slicing:

```
fieldB_z = h5fieldB_z[:]

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

This loads the entire dataset into the main memory of all *engines*. The array elements are distributed along `distaxis=0`.

We can also load a hdf5-group:

```
h5fieldE = pyDive.h5.fromPath("sample.h5", "/fields/fieldE", distaxis=0)
fieldE = h5fieldE[:]
```

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

```
>>> print h5fieldE
VirtualArrayOfStructs<array-type: <class 'pyDive.h5_ndarray.h5_ndarray.h5_ndarray'>, shape: [256, 256
  y -> float32
  x -> float32

>>> print fieldE
VirtualArrayOfStructs<array-type: <class 'pyDive.ndarray.ndarray.ndarray'>, shape: [256, 256]>:
  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.fromPath(h5input, "/fields", distaxis=0)
fields = h5fields[:] # 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[:]` will crash. In this case we want to load the hdf5 data piece by piece. The functions in `pyDive.algorithm` help us doing so:

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

h5input = "sample.h5"

h5fields = pyDive.h5.fromPath(h5input, "/fields", distaxis=0)

def square_fields(npfields):
    return npfields["fieldE/x"]**2 + npfields["fieldE/y"]**2 + npfields["fieldB/z"]**2
```

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

square_fields is called on each *engine* where *npfield* is a structure (`pyDive.arrayOfStructs`) of numpy-arrays 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_ndarrays` and `pyDive.cloned_ndarrays`. If there are `pyDive.h5_ndarrays` it will check whether they fit into the cluster's main memory 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
```

This is valid if *h5fields[::10, ::10]* fits into the cluster's main memory. Note that slicing on a `pyDive.h5_ndarray` always means reading or writing from hdf5 to respectively from memory. So in this case we also could have used the very first version:

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

h5input = "sample.h5"

h5fields = pyDive.h5.fromPath(h5input, "/fields", distaxis=0)
fields = h5fields[::10, ::10]

energy_field = fields["fieldE/x"]**2 + fields["fieldE/y"]**2 + fields["fieldB/z"]**2

total_energy = pyDive.reduce(energy_field, np.add) * 10.0**2
print total_energy
```

But if *h5fields[::10, ::10]* doesn't fit we have to apply the slicing somewhere else in fact at the instanciation of *h5fields*:

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

h5input = "sample.h5"

h5fields = pyDive.h5.fromPath(h5input, "/fields", distaxis=0, window=np.s_[::10, ::10])

def square_fields(npfields):
    return npfields["fieldE/x"]**2 + npfields["fieldE/y"]**2 + npfields["fieldB/z"]**2

total_energy = pyDive.mapReduce(square_fields, np.add, h5fields) * 10.0**2
print total_energy
```

This way the hdf5 data is sliced without involving file i/o.

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(npfield):
    return npfield["x"]**2 + npfield["y"]**2 + npfield["z"]**2
```

```
for step, h5field in pyDive.picongpu.loadAllSteps("/.../simOutput", "fields/FieldE", distaxis=0):
    total_energy = pyDive.mapReduce(square_field, np.add, h5field)

    print step, total_energy
```

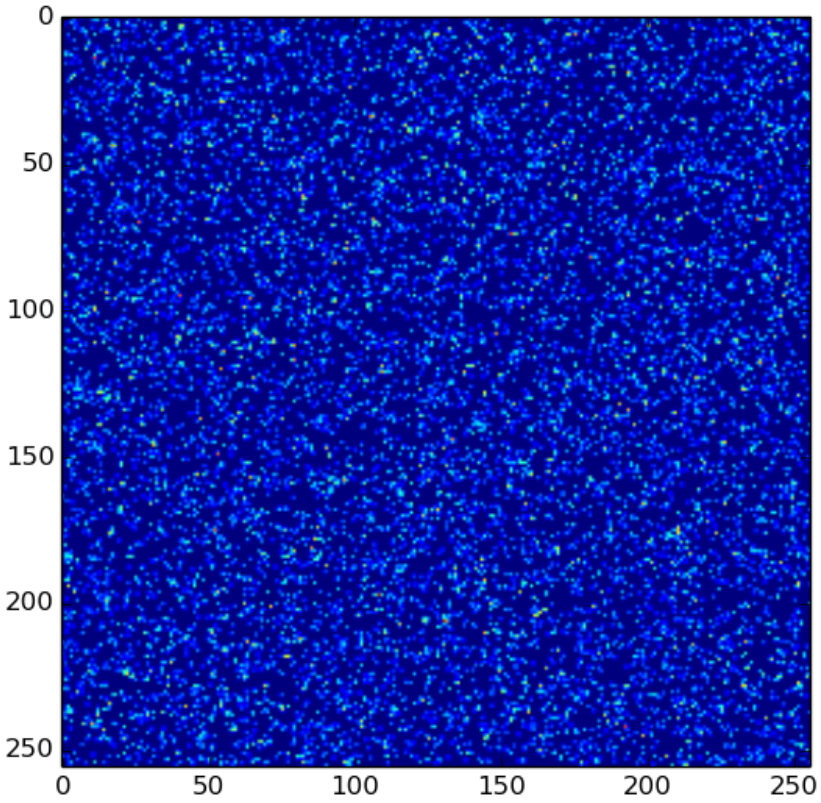
2.2 Example 2: Particle density field

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 basically 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.fromPath(h5input, "/particles", distaxis=0)

def particles2density(particles, density):
    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:

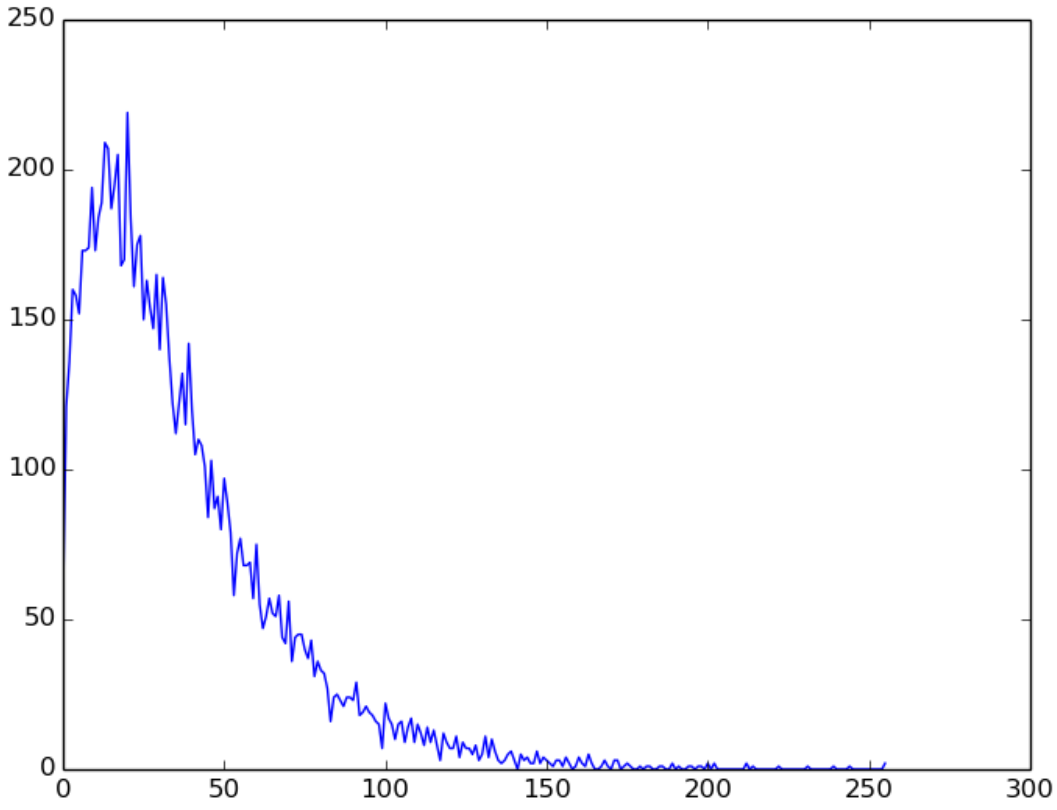
```
@pyDive.map
def particles2density(particles, density):
    ...

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.fromPath(h5input, "/particles/vel", distaxis=0)
```

```
def vel2spectrum(velocities, spectrum, bins):
    mass = 1.0
    energies = 0.5 * mass * (velocities["x"]**2 + velocities["y"]**2 + velocities["z"]**2)

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

pyDive.map(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 Packages

3.1.1 pyDive.ndarray package

Submodules

pyDive.ndarray.ndarray module

class pyDive.ndarray.ndarray.**ndarray**(*shape*, *distaxis=0*, *dtype=<type 'float'>*, *idx_ranges=None*, *targets_in_use=None*, *no_allocation=False*)

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

The implementation is based on *IPython.parallel* and local *numpy-arrays*. The design goal is to forward every *numpy-array* method onto the cluster-wide level. Currently pyDive.ndarray.ndarray.ndarray supports basic arithmetic operations (+ - * / ** //) as well as most of the numpy-*math* functions like sin, cos, abs, sqrt, ... (see pyDive.ndarray.dist_math)

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 *numpy-arrays*.

Every cluster-wide array operation first equalizes the distribution of all involved arrays if necessary.

__init__(*shape*, *distaxis=0*, *dtype=<type 'float'>*, *idx_ranges=None*, *targets_in_use=None*, *no_allocation=False*)

Creates an pyDive.ndarray.ndarray.ndarray instance. This is a low-level method for instanciating an array. Arrays should be constructed using 'empty', 'zeros' or 'array' (see pyDive.ndarray.factories).

Parameters

- **shape** (*tuple of ints*) – size of the array on each axis
- **distaxis** (*int*) – axis on which memory is distributed across the *engines*
- **dtype** (*numpy-dtype*) – datatype of a single data value
- **idx_ranges** (*tuple/list of (int, int)*) – list of (begin, end) pairs indicating the index range the corresponding *engine* (see *targets_in_use*) is associated with
- **targets_in_use** (*tuple/list of ints*) – list of *engine*-ids that share this array.

- **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.

Raises ValueError if just *idx_ranges* is given and *targets_in_use* not or vice versa

If *idx_ranges* and *targets_in_use* are both `None` they will be auto-generated so that the memory is equally distributed across all *engines* at its best. This means that the last engine may get less memory than the others.

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 via MPI.

Parameters **other** (`pyDive.ndarray.ndarray.ndarray`) – target array

Raises

- **AssertionError** – if the shapes of *self* and *other* don't match.
- **AssertionError** – if *self* and *other* are distributed along distinct axes.

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 the local *numpy-arrays* from the *engines*, concatenates them and returns the result.

Returns numpy-array

pyDive.ndarray.factories module

This module holds high-level functions for instanciating *pyDive.ndarrays*.

pyDive.ndarray.factories.**array**(*array_like*, *distaxis=0*)
Return a new *pyDive.ndarray package* from an array-like object.

Parameters

- **array_like** (*array-like*) – Any object exposing the array interface, e.g. numpy-array, python sequence, ...
- **distaxis** (*int*) – axis on which memory is distributed across the *engines*

pyDive.ndarray.factories.**empty**(*shape*, *distaxis=0*, *dtype=<type 'float'>*)
Return a new *pyDive.ndarray package* distributed across all *engines* without initializing elements.

Parameters

- **shape** (*ints*) – shape of the array
- **distaxis** (*int*) – axis on which memory is distributed across the *engines*
- **dtype** (*numpy-dtype*) – datatype of a single data value

pyDive.ndarray.factories.**empty_like**(*a*)
Return a new *pyDive.ndarray package* with the same shape, distribution and type as *a* without initializing elements.

pyDive.ndarray.factories.**hollow**(*shape*, *distaxis=0*, *dtype=<type 'float'>*)

Return a new *pyDive.ndarray package* distributed across all *engines* without allocating a local *numpy-array*.

Parameters

- **shape** (*ints*) – shape of the array
- **distaxis** (*int*) – axis on which memory is distributed across the *engines*
- **dtype** (*numpy-dtype*) – datatype of a single data value

pyDive.ndarray.factories.**hollow_like**(*a*)
Return a new *pyDive.ndarray package* with the same shape, distribution and type as *a* without allocating a local *numpy-array*.

pyDive.ndarray.factories.**ones**(*shape*, *distaxis=0*, *dtype=<type 'float'>*)
Return a new *pyDive.ndarray package* distributed across all *engines* filled with ones.

Parameters

- **shape** (*ints*) – shape of the array
- **distaxis** (*int*) – axis on which memory is distributed across the *engines*
- **dtype** (*numpy-dtype*) – datatype of a single data value

pyDive.ndarray.factories.**ones_like**(*a*)
Return a new *pyDive.ndarray package* with the same shape, distribution and type as *a* filled with ones.

pyDive.ndarray.factories.**zeros**(*shape*, *distaxis=0*, *dtype=<type 'float'>*)
Return a new *pyDive.ndarray package* distributed across all *engines* filled with zeros.

Parameters

- **shape** (*ints*) – shape of the array
- **distaxis** (*int*) – axis on which memory is distributed across the *engines*
- **dtype** (*numpy-dtype*) – datatype of a single data value

pyDive.ndarray.factories.**zeros_like**(*a*)
Return a new *pyDive.ndarray package* with the same shape, distribution and type as *a* filled with zeros.

pyDive.ndarray.dist_math module

Mathematical functions supporting pyDive.ndarray.ndarray.ndarray:

trigonometric:

sin, cos, tan, arcsin, arccos, arctan

hyperbolic:

sinh, cosh, tanh, arcsinh, arccosh, arctanh

rounding:

around, round, rint, fix, floor, ceil, trunc

exponential and logarithmic:

exp, expm1, exp2, log, log10, log2, log1p

misc:

abs, sqrt, maximum, minimum

3.1.2 pyDive.h5_ndarray package

Submodules

pyDive.h5_ndarray.h5_ndarray module

class pyDive.h5_ndarray.h5_ndarray.**h5_ndarray**(*h5_filename*, *dataset_path*, *distaxis=0*, *window=None*)

Represents a single hdf5-dataset like a virtual, cluster-wide array. Data access goes through array slicing where data is written to respectively read from pyDive.ndarray.ndarray objects in parallel using all engines.

Example:

```
h5_data = pyDive.h5.fromPath(<file_path>, "/data/0/fields/FieldB/x", distaxis=0)
data = h5_data[:] # read the entire dataset into engine-memory
data = data**2
h5_data[:] = data # write everything back
```

__init__(*h5_filename*, *dataset_path*, *distaxis=0*, *window=None*)

Creates an h5_ndarray instance. By using this method you may only load a single dataset. If you want to load a *structure* of datasets at once see pyDive.h5_ndarray.factories.fromPath().

Parameters

- **h5_filename** (*str*) – Path of the hdf5-file
- **dataset_path** (*str*) – Path to the dataset within the hdf5-file
- **distaxis** (*int*) – axis on which dataset is to be distributed over during data-access
- **window** (list of slice objects (numpy.s_).) – This param let you specify a sub-part of the array as a virtual container. Example: window=np.s_[:,:,::2]

Notes:

- The dataset's attributes are stored in h5array.attrs.

pyDive.h5_ndarray.factories module

pyDive.h5_ndarray.factories.**fromPath**(*h5_filename*, *datapath*, *distaxis=0*, *window=None*)

Creates a pyDive.h5_ndarray or structure of *pyDive.h5_ndarrays* from a hdf5-dataset respectively a hdf5-group.

Parameters

- **h5_filename** (*str*) – hdf5-filename
- **datapath** (*str*) – path within the hdf5-file to a dataset or a group
- **distaxis** (*int*) – axis on which memory is distributed across the *engines*
- **window** (list of slice objects (*numpy.s_*)) – This param let you specify a sub-part of the array as a virtual container. Example: window=np.s_[:,:,::2]

Returns pyDive.h5_ndarray or structure of pyDive.h5_ndarrays (pyDive.arrayOfStructs)

Notes:

- The dataset's attributes are stored in h5array.attrs.

pyDive.h5_ndarray.h5caching module

pyDive.h5_ndarray.h5caching.**fraction_of_av_mem_used = 0.25**
fraction of the available memory per engine used for caching hdf5 files.

3.1.3 pyDive.cloned_ndarray package

Submodules

pyDive.cloned_ndarray.cloned_ndarray module

class pyDive.cloned_ndarray.cloned_ndarray.**cloned_ndarray**(*shape*, *dtype=<type 'float'>*, *targets_in_use='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'>*, *targets_in_use='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
- **targets_in_use** (*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_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.ndarray package*

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.ndarray package*

3.2 Modules

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"]
```

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*)

Calls *f* on *engine* with local numpy-arrays related to *arrays*. Example:

```
cluster_array = pyDive.ones(shape=[100], distaxis=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 *distaxis* attributes of *pyDive.ndarrays* and *pyDive.h5_ndarrays* do not match

Notes:

- If the hdf5 data exceeds the memory limit (see pyDive.h5_ndarray.h5caching.fraction_of_av_mem_us 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*)

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

```
cluster_array = pyDive.ones(shape=[100], distaxis=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 *distaxis* attributes of *pyDive.ndarrays* and *pyDive.h5_ndarrays* do not match

Notes:

- If the hdf5 data exceeds the memory limit (see pyDive.h5_ndarray.h5caching.fraction_of_av_mem_us 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 (see `pyDive.h5_ndarray.h5caching.fraction_of_av_mem_used`) the data will be read block-wise so that a block fits into memory.

3.2.3 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 at 0x2af19ede5738>*)
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 at 0x2af19ede5738>*)
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.

3.2.4 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*, *window=None*)
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.
- **window** (list of slice objects (*numpy.s_*).) – This param let you specify a sub-part of the array as a virtual container. Example: window=np.s_[:,:,::2]

Returns tuple of timestep and a *pyDive.h5_ndarray* or a structure of pyDive.h5_ndarrays (`pyDive.arrayOfStructs`). 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*, *window=None*)
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.
- **window** (list of slice objects (*numpy.s_*).) – This param let you specify a sub-part of the array as a virtual container. Example: window=np.s_[:,:,::2]

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

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*, *window=None*)
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.
- **window** (list of slice objects (*numpy.s_*).) – This param let you specify a sub-part of the array as a virtual container. Example: window=np.s_[:,:,::2]

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

Notes:

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

3.2.5 pyDive.pyDive module

Make most used functions and modules directly accessable from pyDive.

Functions:

`array`

`arrayOfStructs`

`empty`

`empty_like`

`hollow`

`hollow_like`

`init`

`map`

`mapReduce`

`mesh2particles`

`ones`

`ones_like`

`particles2mesh`

`reduce`

`zeros`

`zeros_like`

Modules:

`IPParallelClient`

`algorithm`

`cloned`

cloned_ndarray

h5

h5_ndarray

mappings

ndarray

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.

p

Symbols

A

C

D

E

F

G

H

L

M

N

O

P

R

S

Z