DistArray Documentation

Release 0.7.0-dev

IPython Development Team and Enthought, Inc.

November 10, 2015

Contents

Think globally, act locally.

DistArray provides general multidimensional NumPy-like distributed arrays to Python. It intends to bring the strengths of NumPy to data-parallel high-performance computing. DistArray has a similar API to NumPy.

DistArray is ready for real-world testing and deployment; however, the project is still evolving rapidly, and we appreciate continued input from the scientific-Python community.

DistArray is for users who

- know and love Python and NumPy,
- want to scale NumPy to larger distributed datasets,
- want to interactively play with distributed data but also
- want to run batch-oriented distributed programs;
- want an easier way to drive and coordinate existing MPI-based codes,
- have a lot of data that may already be distributed,
- want a global view ("think globally") with local control ("act locally"),
- need to tap into existing parallel libraries like Trilinos, PETSc, or Elemental,
- want the interactivity of IPython and the performance of MPI.

DistArray is designed to work with other packages that implement the Distributed Array Protocol.

CHAPTER 1

Getting Started

To see some examples of what DistArray can do, check out our IPython notebooks on nbviewer (also in the `examples` directory of the DistArray source).

- DistArray Features
- Seismic Volume
- Julia Set
- Gaussian Elimination

CHAPTER 2

Overview

NumPy is at the foundation of the scientific Python stack for good reason: NumPy arrays are easy to use, they have many powerful features like ufuncs, slicing, and broadcasting, and they work easily with external libraries.

As data sets grow and parallel hardware becomes more widely available, wouldn't it be great if NumPy easily supported parallel execution, without losing its nice interface in a miasma of low-level parallel coordination? What would that look like?

What we want is transparent distribution of NumPy arrays over the CPU, cluster, and supercomputer. We want to interact with distributed NumPy arrays the way we think about them and get the benefit of all that parallelism. We also want to be able to drop down a level to control what's going on at the data-local level when performance demands it.

Such a NumPy opens doors to providing a high-level NumPy-like interface to distributed libraries like Trilinos, PETSc, Global Arrays, Elemental, and ScaLAPACK, among others.

All this coordination has overhead and is at risk of becoming a performance bottleneck. This NumPy will need a way to allow direct execution at a data-local level. We will also need a way to communicate directly between local processes when needed, rather than doing everything at a global level.

This distributed NumPy should be a good citizen and work easily with regular NumPy arrays, with MPI, with IPython parallel, and with external distributed algorithms.

DistArray is our vision of what distributed NumPy can be. It brings the best parts of NumPy to data-parallel computing. We want to *think globally* about our arrays, interacting with them as if they are just really big NumPy arrays, all the while *acting locally* on them for performance and control.

CHAPTER 3

Installation

DistArray requires the following Python libraries:

- numpy,
- ipyparallel, and
- mpi4py.

Optionally, DistArray can make use of:

- h5py built against a parallel-enabled build of HDF5 (for HDF5 IO), and
- matplotlib (for making plots of DistArray distributions).

If you have the above, you should be able to install DistArray with:

```
python setup.py install
```

or:

```
pip install distarray
```

CHAPTER 4

Experimental quickstart scripts

Alternatively, we have experimental installation scripts in the `quickstart` directory of the root of this source tree. Given a Canopy or Anaconda installation and a couple of other prerequisites, these scripts attempt to install DistArray and its dependencies for you. See the readme files in that directory for more information.

CHAPTER 5

Testing Your Installation

To test your installation, you will first need to start an IPython.parallel cluster with MPI enabled. The easist way is to use the `dacluster` command that comes with DistArray:

```
dacluster start
```

See `dacluster`'s help for more:

```
dacluster --help
```

You should then be able to run all the tests from the DistArray source directory with:

```
make test
```

If you've installed DistArray with `python setup.py develop`, you should be able to run the tests from anywhere with:

```
python -m distarray.run_tests
```

CHAPTER 6

Building the docs

Dependencies to build the documentation:

- Sphinx >= 1.3
- sphinxcontrib.programoutput

If you have the dependencies listed above, and you want to build the documentation (also available at http://distarray.readthedocs.org), navigate to the `docs/sphinx` subdirectory of the DistArray source and use the Makefile there.

For example, to build the html documentation:

```
make html
```

from the `docs` directory.

Try:

```
make help
```

for more options.

CHAPTER 7

History

DistArray was started by Brian Granger in 2008 and is currently being developed by Enthought in partnership with Bill Spotz from Sandia's (Py)Trilinos project and Brian Granger and Min RK from the IPython project.

CHAPTER 8

Contact Us

If you have questions or would like to contribute, contact us

- on the DistArray mailing list: distarray@googlegroups.com, or
- through the DistArray GitHub repo: https://github.com/enthought/distarray (for bug reports and pull requests).

CHAPTER 9

Other Documentation

9.1 DistArray API Reference

9.1.1 `distarray` Package

DistArray: Distributed NumPy-like arrays

Documentation is available in docstrings and online at http://distarray.readthedocs.org.

Check out the `examples` directory in the source distribution for several example modules and IPython notebooks using DistArray.

9.1.2 `__version__` Module

Version information for the DistArray package.

9.1.3 `error` Module

Exception classes for DistArray errors.

exception `distarray.error.`**`ContextError`**
Bases: *distarray.error.DistArrayError*

Exception class when a unique Context cannot be found.

exception `distarray.error.`**`DistArrayError`**
Bases: `exceptions.Exception`

Base exception class for DistArray errors.

exception `distarray.error.`**`DistributionError`**
Bases: *distarray.error.DistArrayError*

Exception class when inconsistent distributions are used.

exception `distarray.error.`**`InvalidCommSizeError`**
Bases: *distarray.error.MPIDistArrayError*

Exception class when a requested communicator is too large.

exception distarray.error.**InvalidRankError**
Bases: *distarray.error.MPIDistArrayError*

Exception class when an invalid rank is used in a communicator.

exception distarray.error.**MPIDistArrayError**
Bases: *distarray.error.DistArrayError*

Base exception class for MPI distribution errors.

9.1.4 metadata_utils Module

Utility functions for dealing with DistArray metadata.

exception distarray.metadata_utils.**GridShapeError**
Bases: exceptions.Exception

Exception class when it is not possible to distribute the processes over the number of dimensions.

exception distarray.metadata_utils.**InvalidGridShapeError**
Bases: exceptions.Exception

Exception class when the grid shape is incompatible with the distribution or communicator.

distarray.metadata_utils.**arg_kwarg_proxy_converter**(*args*, *kwargs*, *module_name='__main__'*)

distarray.metadata_utils.**block_cyclic_size**(*dim_data*)
Get a size from a block-cyclic dim_data.

distarray.metadata_utils.**block_size**(*dim_data*)
Get a size from a block distributed dim_data.

distarray.metadata_utils.**c_or_bc_chooser**(*dim_data*)
Get a size from a cyclic or block-cyclic dim_data.

distarray.metadata_utils.**check_grid_shape_postconditions**(*grid_shape*, *shape*, *dist*, *comm_size*)
Check grid_shape for reasonableness after creating it.

distarray.metadata_utils.**check_grid_shape_preconditions**(*shape*, *dist*, *comm_size*)
Verify various distarray parameters are correct before making a grid_shape.

distarray.metadata_utils.**condense**(*intervals*)

distarray.metadata_utils.**cyclic_size**(*dim_data*)
Get a size from a cyclic dim_data.

distarray.metadata_utils.**distribute_block_indices**(*dd*)
Fill in *start* and *stop* in dim dict *dd*.

distarray.metadata_utils.**distribute_cyclic_indices**(*dd*)
Fill in *start* in dim dict *dd*.

distarray.metadata_utils.**distribute_indices**(*dd*)
Fill in index related keys in dim dict *dd*.

distarray.metadata_utils.**make_grid_shape**(*shape*, *dist*, *comm_size*)
Generate a *grid_shape* from *shape* tuple and *dist* tuple.

Does not assume that *dim_data* has *proc_grid_size* set for each dimension.

Attempts to allocate processes optimally for distributed dimensions.

Parameters

- **shape** (*tuple of int*) – The global shape of the array.
- **dist** (*tuple of str*) – dist_type character per dimension.
- **comm_size** (*int*) – Total number of processes to distribute.

Returns dist_grid_shape

Return type tuple of int

Raises *GridShapeError* – if not possible to distribute *comm_size* processes over number of dimensions.

distarray.metadata_utils.**ndim_from_flat**(*flat*, *strides*)

distarray.metadata_utils.**non_dist_size**(*dim_data*)
Get a size from a nondistributed dim_data.

distarray.metadata_utils.**normalize_dim_dict**(*dd*)
Fill out some degenerate dim_dicts.

distarray.metadata_utils.**normalize_dist**(*dist*, *ndim*)
Return a tuple containing dist-type for each dimension.

Parameters

- **dist** (*str, list, tuple, or dict*) –
- **ndim** (*int*) –

Returns Contains string distribution type for each dim.

Return type tuple of str

Examples

```
>>> normalize_dist({0: 'b', 3: 'c'}, 4)
('b', 'n', 'n', 'c')
```

distarray.metadata_utils.**normalize_grid_shape**(*grid_shape*, *shape*, *dist*, *comm_size*)
Adds 1s to grid_shape so it has *ndims* dimensions. Validates *grid_shape* tuple against the *dist* tuple and *comm_size*.

distarray.metadata_utils.**normalize_reduction_axes**(*axes*, *ndim*)

distarray.metadata_utils.**positivify**(*index*, *size*)
Check that an index is within bounds and return a positive version.

Parameters

- **index** (*Integral or slice*) –
- **size** (*Integral*) –

Raises IndexError – for out-of-bounds indices

distarray.metadata_utils.**sanitize_indices**(*indices*, *ndim=None*, *shape=None*)
Classify and sanitize *indices*.

•Wrap naked Integral, slice, or Ellipsis indices into tuples

•Classify result as 'value' or 'view'

•Expand *Ellipsis* objects to slices

- If the length of the tuple-ized *indices* is < ndim (and it's provided), add slice(None)'s to indices until *indices* is ndim long
- If *shape* is provided, call *positivify* on the indices

Raises

- `TypeError` – If *indices* is other than Integral, slice or a Sequence of these
- `IndexError` – If len(indices) > ndim

Returns

Return type 2-tuple of (str, n-tuple of slices and Integral values)

`distarray.metadata_utils.`**`shapes_from_dim_data_per_rank`**(*ddpr*)
Given a dim_data_per_rank object, return the shapes of the localarrays. This requires no communication.

`distarray.metadata_utils.`**`size_chooser`**(*dist_type*)
Get a function from a dist_type.

`distarray.metadata_utils.`**`size_from_dim_data`**(*dim_data*)
Get a size from a dim_data.

`distarray.metadata_utils.`**`strides_from_shape`**(*shape*)

`distarray.metadata_utils.`**`tuple_intersection`**(*t0*, *t1*)
Compute intersection of a (start, stop, step) and a (start, stop) tuple.

Assumes all values are positive.

Parameters

- **`t0`** (*2-tuple or 3-tuple*) – Tuple of (start, stop, [step]) representing an index range
- **`t1`** (*2-tuple*) – Tuple of (start, stop) representing an index range

Returns A tightly bounded interval.

Return type 3-tuple or None

`distarray.metadata_utils.`**`unstructured_size`**(*dim_data*)
Get a size from an unstructured dim_data.

9.1.5 `mpi_engine` Module

The engine_loop function and utilities necessary for it.

class `distarray.mpi_engine.`**`Engine`**
Bases: `object`

`INTERCOMM` **= None**

`builtin_call`(*msg*)

`delete`(*msg*)

`engine_make_targets_comm`(*msg*)

`execute`(*msg*)

`free_comm`(*msg*)

`func_call`(*msg*)

`is_engine`()

kill(*msg*)
Break out of the engine loop.

parse_msg(*msg*)

pull(*msg*)

push(*msg*)

9.1.6 mpionly_utils Module

Utilities for running Distarray in MPI mode.

distarray.mpionly_utils.**get_comm_world**()

distarray.mpionly_utils.**get_world_rank**()

distarray.mpionly_utils.**initial_comm_setup**()
Setup client and engine intracomm, and intercomm.

distarray.mpionly_utils.**is_solo_mpi_process**()

distarray.mpionly_utils.**make_targets_comm**(*targets*)

distarray.mpionly_utils.**push_function**(*context*, *key*, *func*, *targets=None*)

9.1.7 run_tests Module

Functions for running DistArray tests.

distarray.run_tests.**test**()
Run all DistArray tests.

9.1.8 testing Module

Functions used for tests.

class distarray.testing.**BaseContextTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Base test class for test cases that use a Context.

Overload the *ntargets* class attribute to change the default number of engines required. A *cls.context* object will be created with *targets=range(cls.ntargets)*. Tests will be skipped if there are too few targets.

ntargets
int or 'any', default=4

If an int, indicates how many engines are required for this test to run. If the string 'any', indicates that any number of engines may be used with this test.

ntargets = 4

classmethod setUpClass()

classmethod tearDownClass()

class distarray.testing.**CommNullPasser**
Bases: type

Metaclass.

Applies the *comm_null_passes* decorator to every method on a generated class.

class distarray.testing.**DefaultContextTestCase**(*methodName='runTest'*)
Bases: *distarray.testing.BaseContextTestCase*

classmethod make_context(*targets=None*)

class distarray.testing.**IPythonContextTestCase**(*methodName='runTest'*)
Bases: *distarray.testing.BaseContextTestCase*

classmethod make_context(*targets=None*)

classmethod setUpClass()

class distarray.testing.**MPIContextTestCase**(*methodName='runTest'*)
Bases: *distarray.testing.BaseContextTestCase*

classmethod make_context(*targets=None*)

class distarray.testing.**ParallelTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Base test class for fully distributed and client-less test cases.

Overload the *comm_size* class attribute to change the default number of processes required.

comm_size
int, default=4

Indicates how many MPI processes are required for this test to run. If fewer than *comm_size* are available, the test will be skipped.

comm_size = 4

classmethod setUpClass()

classmethod tearDownClass()

distarray.testing.**assert_localarrays_allclose**(*l0*, *l1*, *check_dtype=False*, *rtol=1e-07*, *atol=0*)
Call np.testing.assert_allclose on *l0* and *l1*.

Also, check that LocalArray properties are equal.

distarray.testing.**assert_localarrays_equal**(*l0*, *l1*, *check_dtype=False*)
Call np.testing.assert_equal on *l0* and *l1*.

Also, check that LocalArray properties are equal.

distarray.testing.**check_targets**(*required*, *available*)
If available < required, raise a SkipTest with a nice error message.

distarray.testing.**comm_null_passes**(*fn*)
Decorator. If *self.comm* is COMM_NULL, pass.

This allows our tests to pass on processes that have nothing to do.

distarray.testing.**import_or_skip**(*name*)
Try importing *name*, raise SkipTest on failure.

Parameters **name** (*str*) – Module name to try to import.

Returns **module** – Module object imported by importlib.

Return type module object

Raises unittest.SkipTest – If the attempted import raises an ImportError.

Examples

```
>>> h5py = import_or_skip('h5py')
>>> h5py.get_config()
<h5py.h5.H5PYConfig at 0x103dd5a78>
```

distarray.testing.**raise_typeerror**(*fn*)
Decorator for protocol validator functions.

These functions return (success, err_msg), but sometimes we would rather have an exception.

distarray.testing.**temp_filepath**(*extension=''*)
Return a randomly generated filename.

This filename is appended to the directory path returned by *tempfile.gettempdir()* and has *extension* appended to it.

9.1.9 `utils` Module

Utilities.

distarray.utils.**all_equal**(*iterable*)
Return True if all elements in *iterable* are equal.

Also returns True if iterable is empty.

class distarray.utils.**count_round_trips**(*client*)
Bases: object

Context manager for counting the number of roundtrips between a IPython client and controller.

Usage:

```
>>> with count_round_trips(client) as r:
...     send_42_messages()
```

```
>>> r.count
42
```

update_count()

distarray.utils.**distarray_random_getstate**()
Get the state of the global random number generator.

distarray.utils.**distarray_random_setstate**(*state*)
Set the state of the global random number generator.

distarray.utils.**divisors_minmax**(*n*, *dmin*, *dmax*)
Find the divisors of n in the interval (dmin,dmax].

distarray.utils.**flatten**(*seq*, *to_expand=<function list_or_tuple>*)
Flatten a nested sequence.

distarray.utils.**get_from_dotted_name**(*dotted_name*)

distarray.utils.**has_exactly_one**(*iterable*)
Does *iterable* have exactly one non-None element?

distarray.utils.**list_or_tuple**(*seq*)
Is the object either a list or a tuple?

distarray.utils.**mirror_sort**(*seq*, *ref_seq*)
Sort *seq* into the order that *ref_seq* is in.

```
>>> mirror_sort(range(5),[1,5,2,4,3])
[0, 4, 1, 3, 2]
```

distarray.utils.**mult_partitions**(*n*, *s*)
Compute the multiplicative partitions of n of size s

```
>>> mult_partitions(52,3)
[(2, 2, 13)]
>>> mult_partitions(52,2)
[(2, 26), (4, 13)]
```

distarray.utils.**mult_partitions_recurs**(*n*, *s*, *pd=0*)

distarray.utils.**multi_for**(*iterables*)

distarray.utils.**nonce**()

distarray.utils.**remove_elements**(*to_remove*, *seq*)
Return a list, with the elements with specified indices removed.

Parameters

- **to_remove** (*iterable*) – Indices of elements in list to remove
- **seq** (*iterable*) – Elements in the list.

Returns

Return type List with the specified indices removed.

distarray.utils.**set_from_dotted_name**(*name*, *val*)

distarray.utils.**slice_intersection**(*s1*, *s2*)
Compute a slice that represents the intersection of two slices.

Currently only implemented for steps of size 1.

Parameters **s2** (*s1,*) –

Returns

Return type slice object

distarray.utils.**uid**()
Get a unique name for a distarray object.

9.1.10 Subpackages

apps Package

apps Package

Scripts for use with the DistArray package.

dacluster Module

Start, stop and manage a IPython.parallel cluster. *dacluster* can take all the commands IPython's *ipcluster* can, and a few extras that are distarray specific.

distarray.apps.dacluster.**clear**(***kwargs*)
Removes all distarray-related modules from engines' sys.modules.

distarray.apps.dacluster.**main**()
Main function for dacluster utility.

Either start, stop, restart, or clear is called depending on the command line arguments.

distarray.apps.dacluster.**restart**(*n=4*, *engines=None*, ***kwargs*)
Convenient way to restart an ipcluster.

distarray.apps.dacluster.**start**(*n=4*, *engines=None*, ***kwargs*)
Convenient way to start an ipcluster for testing.

Doesn't exit until the ipcluster prints a success message.

distarray.apps.dacluster.**stop**(***kwargs*)
Convenient way to stop an ipcluster.

```
Traceback (most recent call last):
  File "/usr/lib/python2.7/runpy.py", line 162, in _run_module_as_main
    "__main__", fname, loader, pkg_name)
  File "/usr/lib/python2.7/runpy.py", line 72, in _run_code
    exec code in run_globals
  File "/home/docs/checkouts/readthedocs.org/user_builds/distarray/envs/master/lib/python2.7/site-pac
    from distarray.globalapi.ipython_cleanup import clear_all
  File "/home/docs/checkouts/readthedocs.org/user_builds/distarray/envs/master/local/lib/python2.7/si
    from distarray.globalapi.distarray import DistArray
  File "/home/docs/checkouts/readthedocs.org/user_builds/distarray/envs/master/local/lib/python2.7/si
    import numpy as np
ImportError: No module named numpy
```

engine Module

Script for facilitating MPI-only mode.

Starts an MPI-process-based engine.

```
Traceback (most recent call last):
  File "/usr/lib/python2.7/runpy.py", line 162, in _run_module_as_main
    "__main__", fname, loader, pkg_name)
  File "/usr/lib/python2.7/runpy.py", line 72, in _run_code
    exec code in run_globals
  File "/home/docs/checkouts/readthedocs.org/user_builds/distarray/envs/master/lib/python2.7/site-pac
    from distarray.mpi_engine import Engine
  File "/home/docs/checkouts/readthedocs.org/user_builds/distarray/envs/master/local/lib/python2.7/si
    from distarray.metadata_utils import arg_kwarg_proxy_converter
  File "/home/docs/checkouts/readthedocs.org/user_builds/distarray/envs/master/local/lib/python2.7/si
    import numpy
ImportError: No module named numpy
```

globalapi Package

globalapi Package

Modules dealing with the global index-space view of *DistArray*s.

In other words, the view from the client.

`context` Module

Context objects contain the information required for *DistArray*s to communicate with *LocalArray*s.

class `distarray.globalapi.context.`**`BaseContext`**
Bases: `object`

Context objects manage the setup and communication of the worker processes for DistArray objects. A DistArray object has a context, and contexts have an MPI intracommunicator that they use to communicate with worker processes.

Typically there is just one context object that uses all processes, although it is possible to have more than one context with a different selection of engines.

`allclose`(*a*, *b*, *rtol=1e-05*, *atol=1e-08*)

`apply`(*func*, *args=None*, *kwargs=None*, *targets=None*, *autoproxyize=False*)

`cleanup`()

`close`()

`delete_key`(*key*, *targets=None*)
Delete the specific key from all the engines.

`empty`(*shape_or_dist*, *dtype=<type 'float'>*)
Create an empty Distarray.

Parameters

- **`shape_or_dist`** (*shape tuple or Distribution object*) –
- **`dtype`** (*NumPy dtype, optional (default float)*) –

Returns A DistArray distributed as specified, with uninitialized values.

Return type *DistArray*

`fromarray`(*arr*, *distribution=None*)
Create a DistArray from an ndarray.

Parameters **`distribution`** (*Distribution object, optional*) – If a Distribution object is not provided, one is created with *Distribution(arr.shape)*.

Returns A DistArray distributed as specified, using the values and dtype from *arr*.

Return type *DistArray*

`fromfunction`(*function*, *shape*, ***kwargs*)
Create a DistArray from a function over global indices.

Unlike numpy's *fromfunction*, the result of distarray's *fromfunction* is restricted to the same Distribution as the index array generated from *shape*.

See numpy.fromfunction for more details.

`fromndarray`(*arr*, *distribution=None*)
Create a DistArray from an ndarray.

Parameters **`distribution`** (*Distribution object, optional*) – If a Distribution object is not provided, one is created with *Distribution(arr.shape)*.

Returns A DistArray distributed as specified, using the values and dtype from *arr*.

Return type *DistArray*

load_dnpy(*name*)
Load a distributed array from `.dnpy` files.

The `.dnpy` file format is a binary format inspired by NumPy's `.npy` format. The header of a particular `.dnpy` file contains information about which portion of a DistArray is saved in it (using the metadata outlined in the Distributed Array Protocol), and the data portion contains the output of NumPy's *save* function for the local array data. See the module docstring for *distarray.localapi.format* for full details.

Parameters **name** (*str or list of str*) – If a str, this is used as the prefix for the filename used by each engine. Each engine will load a file named `<name>_<rank>.dnpy`. If a list of str, each engine will use the name at the index corresponding to its rank. An exception is raised if the length of this list is not the same as the context's communicator's size.

Returns **result** – A DistArray encapsulating the file loaded on each engine.

Return type *DistArray*

Raises `TypeError` – If *name* is an iterable whose length is different from the context's communicator's size.

See also:

save_dnpy() Saving files to load with with load_dnpy.

load_hdf5(*filename*, *distribution*, *key='buffer'*)
Load a DistArray from a dataset in an `.hdf5` file.

Parameters

- **filename** (*str*) – Filename to load.
- **distribution** (*Distribution object*) –
- **key** (*str, optional*) – The identifier for the group to load the DistArray from (the default is 'buffer').

Returns **result** – A DistArray encapsulating the file loaded.

Return type *DistArray*

load_npy(*filename*, *distribution*)
Load a DistArray from a dataset in a `.npy` file.

Parameters

- **filename** (*str*) – Filename to load.
- **distribution** (*Distribution object*) –

Returns **result** – A DistArray encapsulating the file loaded.

Return type *DistArray*

make_subcomm(*new_targets*)

ones(*shape_or_dist*, *dtype=<type 'float'>*)
Create a Distarray filled with ones.

Parameters

- **shape_or_dist** (*shape tuple or Distribution object*) –
- **dtype** (*NumPy dtype, optional (default float)*) –

Returns A DistArray distributed as specified, filled with ones.

Return type *DistArray*

push_function(*key*, *func*)

register(*func*)
Associate a function with this Context. Allows access to the local process and local data associated with each DistArray.

After registering a function with a context, the function can be called as `context.func(...)`. Doing so will call the function locally on target processes determined from the arguments passed in using `Context.apply(...)`. The function can take non-proxied Python objects, DistArrays, or other proxied objects as arguments. Non-proxied Python objects will be broadcasted to all local processes; proxied objects will be dereferenced before calling the function on the local process.

save_dnpy(*name*, *da*)
Save a distributed array to files in the `.dnpy` format.

The `.dnpy` file format is a binary format inspired by NumPy's `.npy` format. The header of a particular `.dnpy` file contains information about which portion of a DistArray is saved in it (using the metadata outlined in the Distributed Array Protocol), and the data portion contains the output of NumPy's *save* function for the local array data. See the module docstring for *distarray.localapi.format* for full details.

Parameters

- **name** (*str or list of str*) – If a str, this is used as the prefix for the filename used by each engine. Each engine will save a file named `<name>_<rank>.dnpy`. If a list of str, each engine will use the name at the index corresponding to its rank. An exception is raised if the length of this list is not the same as the context's communicator's size.
- **da** (DistArray) – Array to save to files.

Raises `TypeError` – If *name* is an sequence whose length is different from the context's communicator's size.

See also:

load_dnpy() Loading files saved with save_dnpy.

save_hdf5(*filename*, *da*, *key='buffer'*, *mode='a'*)
Save a DistArray to a dataset in an `.hdf5` file.

Parameters

- **filename** (*str*) – Name of file to write to.
- **da** (DistArray) – Array to save to a file.
- **key** (*str, optional*) – The identifier for the group to save the DistArray to (the default is 'buffer').
- **mode** (*optional, {'w', 'w-', 'a'}, default 'a'*) –

  'w' Create file, truncate if exists

  'w-' Create file, fail if exists

  'a' Read/write if exists, create otherwise (default)

zeros(*shape_or_dist*, *dtype=<type 'float'>*)
Create a Distarray filled with zeros.

Parameters

- **shape_or_dist** (*shape tuple or Distribution object*) –

- **dtype** (*NumPy dtype, optional (default float)*) –

Returns A DistArray distributed as specified, filled with zeros.

Return type *DistArray*

distarray.globalapi.context.**Context**(**args*, ***kwargs*)

exception distarray.globalapi.context.**ContextCreationError**
Bases: exceptions.RuntimeError

class distarray.globalapi.context.**IPythonContext**(*client=None*, *targets=None*)
Bases: *distarray.globalapi.context.BaseContext*

Context class that uses IPython.parallel.

See the docstring for *BaseContext* for more information about Contexts.

See also:

BaseContext

apply(*func*, *args=None*, *kwargs=None*, *targets=None*, *autoproxyize=False*)
Analogous to IPython.parallel.view.apply_sync

Parameters

- **func** (*function*) –
- **args** (*tuple*) – positional arguments to func
- **kwargs** (*dict*) – key word arguments to func
- **targets** (*sequence of integers*) – engines func is to be run on.
- **autoproxyize** (*bool, default False*) – If True, implicitly return a Proxy object from the function.

Returns

Return type return a list of the results on the each engine.

cleanup()
Delete keys that this context created from all the engines.

close()

make_subcomm(*new_targets*)

push_function(*key*, *func*, *targets=None*)

class distarray.globalapi.context.**MPIContext**(*targets=None*)
Bases: *distarray.globalapi.context.BaseContext*

Context class that uses MPI only (no IPython.parallel).

See the docstring for *BaseContext* for more information about Contexts.

See also:

BaseContext

INTERCOMM = None

apply(*func*, *args=None*, *kwargs=None*, *targets=None*, *autoproxyize=False*)
Analogous to IPython.parallel.view.apply_sync

Parameters

- **func** (*function*) –
- **args** (*tuple*) – positional arguments to func
- **kwargs** (*dict*) – keyword arguments to func
- **targets** (*sequence of integers*) – engines func is to be run on.
- **autoproxyize** (*bool, default False*) – If True, implicitly return a Proxy object from the function.

Returns result from each engine.

Return type list

cleanup()
Delete keys that this context created from all the engines.

close()

delete_key(*key*, *targets=None*)

make_subcomm(*targets*)

push_function(*key*, *func*, *targets=None*)

distarray Module

The *DistArray* data structure.

DistArray objects are proxies for collections of *LocalArray* objects. They are meant to roughly emulate NumPy *ndarray*s.

class distarray.globalapi.distarray.**DistArray**(*distribution*, *dtype=<type 'float'>*)
Bases: object

context

dist

distribute_as(*shape_or_dist*)
Redistributes this DistArray, returning a new DistArray with the same data and corresponding distribution.

Parameters **shape_or_dist** (*shape tuple or Distribution object.*) – Distribution for the new DistArray. The new distribution must have the same number of items as this distarray. The global shape and targets may be different. If shape tuple, immediately converted to a Distribution object with default parameters.

Returns A new DistArray distributed according to *dist*.

Return type *DistArray*

Note: Currently implemented for block and non-distributed maps only.

dtype

fill(*value*)

classmethod **from_localarrays**(*key*, *context=None*, *targets=None*, *distribution=None*, *dtype=None*)
The caller has already created the LocalArray objects. *key* is their name on the engines. This classmethod creates a DistArray that refers to these LocalArrays.

Either a *context* or a *distribution* must also be provided. If *context* is provided, a `dim_data_per_rank` will be pulled from the existing `LocalArray`s and a `Distribution` will be created from it. If *distribution* is provided, it should accurately reflect the distribution of the existing `LocalArray`s.

If *dtype* is not provided, it will be fetched from the engines.

get_localarrays()
Pull the LocalArray objects from the engines.

Returns one localarray per process

Return type list of localarrays

get_ndarrays()
Pull the local ndarrays from the engines.

Returns one ndarray per process

Return type list of ndarrays

global_size

grid_shape

itemsize

localshapes()

max(*axis=None*, *dtype=None*, *out=None*)
Return the maximum of array elements over the given axis.

mean(*axis=None*, *dtype=<type 'float'>*, *out=None*)
Return the mean of array elements over the given axis.

min(*axis=None*, *dtype=None*, *out=None*)
Return the minimum of array elements over the given axis.

nbytes

ndim

shape

std(*axis=None*, *dtype=<type 'float'>*, *out=None*)
Return the standard deviation of array elements over the given axis.

sum(*axis=None*, *dtype=None*, *out=None*)
Return the sum of array elements over the given axis.

targets

toarray()
Returns the distributed array as an ndarray.

tondarray()
Returns the distributed array as an ndarray.

var(*axis=None*, *dtype=<type 'float'>*, *out=None*)
Return the variance of array elements over the given axis.

view(*dtype=None*)
New view of array with the same data.

Parameters **dtype** (*numpy dtype, optional*) – Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, results in the view having the same data-type as the original array.

Returns A view on the original DistArray, optionally with the underlying memory interpreted as a different dtype.

Return type *DistArray*

Note: No redistribution is done. The sizes of all *LocalArray*s must be compatible with the new view.

`functions` Module

Distributed ufuncs for *DistArray*s.

`distarray.globalapi.functions.`**`absolute`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`arccos`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`arccosh`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`arcsin`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`arcsinh`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`arctan`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`arctanh`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`conjugate`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`cos`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`cosh`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`exp`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`expm1`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`invert`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`log`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`log10`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`log1p`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`negative`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`reciprocal`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`rint`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`sign`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`sin`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`sinh`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`sqrt`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`square`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`tan`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`tanh`**(*a*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`add`**(*a*, *b*, **args*, ***kwargs*)

`distarray.globalapi.functions.`**`arctan2`**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**bitwise_and**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**bitwise_or**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**bitwise_xor**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**divide**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**floor_divide**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**fmod**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**hypot**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**left_shift**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**mod**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**multiply**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**power**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**remainder**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**right_shift**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**subtract**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**true_divide**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**less**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**less_equal**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**equal**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**not_equal**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**greater**(*a*, *b*, **args*, ***kwargs*)

distarray.globalapi.functions.**greater_equal**(*a*, *b*, **args*, ***kwargs*)

ipython_cleanup Module

Functions for cleaning up *DistArray* objects from IPython parallel engines.

distarray.globalapi.ipython_cleanup.**cleanup**(*view*, *module_name*, *prefix*)
Delete Context object with the given name from the given module

distarray.globalapi.ipython_cleanup.**cleanup_all**(*module_name*, *prefix*)
Connects to all engines and runs `cleanup()` on them.

distarray.globalapi.ipython_cleanup.**clear**(*view*)
Removes all distarray-related modules from engines' sys.modules.

distarray.globalapi.ipython_cleanup.**clear_all**()

distarray.globalapi.ipython_cleanup.**get_local_keys**(*view*, *prefix*)
Returns a dictionary of keyname -> target_list mapping for all names that start with `prefix` on engines in `view`.

ipython_utils Module

The single IPython entry point.

maps Module

Distribution class and auxiliary ClientMap classes.

The Distribution is a multi-dimensional map class that manages the one-dimensional maps for each DistArray dimension. The Distribution class represents the *distribution* information for a distributed array, independent of the distributed array's *data*. Distributions allow DistArrays to reduce overall communication when indexing and slicing by determining which processes own (or may possibly own) the indices in question. Two DistArray objects can share the same Distribution if they have the exact same distribution.

The one-dimensional ClientMap classes keep track of which process owns which index in that dimension. This class has several subclasses for specific distribution types, including *BlockMap*, *CyclicMap*, *NoDistMap*, and *UnstructuredMap*.

class distarray.globalapi.maps.**BlockCyclicMap**(*size*, *grid_size*, *block_size=1*)
Bases: *distarray.globalapi.maps.MapBase*

dist = 'c'

classmethod from_axis_dim_dicts(*axis_dim_dicts*)

classmethod from_global_dim_dict(*glb_dim_dict*)

get_dimdicts()

index_owners(*idx*)

is_compatible(*other*)

class distarray.globalapi.maps.**BlockMap**(*size*, *grid_size*, *bounds=None*, *comm_padding=None*, *boundary_padding=None*)
Bases: *distarray.globalapi.maps.MapBase*

dist = 'b'

classmethod from_axis_dim_dicts(*axis_dim_dicts*)

classmethod from_global_dim_dict(*glb_dim_dict*)

get_dimdicts()

index_owners(*idx*)

is_compatible(*other*)

slice(*idx*)
Make a new Map from a slice.

slice_owners(*idx*)

view(*new_dimsize*)
Scale this map for the *view* method.

class distarray.globalapi.maps.**Distribution**
Bases: object

Governs the mapping between global indices and process ranks for multi-dimensional objects.

comm

comm_union(**dists*)
Make a communicator that includes the union of all targets in *dists*.

Parameters **dists** (*sequence of distribution objects.*) –

Returns First element is encompassing communicator proxy; second is a sequence of all targets in *dists*.

Return type tuple

classmethod **from_dim_data_per_rank**(*context*, *dim_data_per_rank*, *targets=None*)

Create a Distribution from a sequence of *dim_data* tuples.

Parameters

- **context** (*Context object*) –
- **dim_data_per_rank** (*Sequence of dim_data tuples, one per rank*) – See the "Distributed Array Protocol" for a description of dim_data tuples.
- **targets** (*Sequence of int, optional*) – Sequence of engine target numbers. Default: all available

Returns

Return type *Distribution*

classmethod **from_global_dim_data**(*context*, *global_dim_data*, *targets=None*)

Make a Distribution from a global_dim_data structure.

Parameters

- **context** (*Context object*) –
- **global_dim_data** (*tuple of dict*) – A global dimension dictionary per dimension. See following *Note* section.
- **targets** (*Sequence of int, optional*) – Sequence of engine target numbers. Default: all available

Returns

Return type *Distribution*

Note: The *global_dim_data* tuple is a simple, straightforward data structure that allows full control over all aspects of a DistArray's distribution information. It does not contain any of the array's *data*, only the *metadata* needed to specify how the array is to be distributed. Each dimension of the array is represented by corresponding dictionary in the tuple, one per dimension. All dictionaries have a *dist_type* key that specifies whether the array is block, cyclic, or unstructured. The other keys in the dictionary are dependent on the *dist_type* key.

Block

- `dist_type` is `'b'`.
- `bounds` is a sequence of integers, at least two elements.

  The `bounds` sequence always starts with 0 and ends with the global `size` of the array. The other elements indicate the local array global index boundaries, such that successive pairs of elements from `bounds` indicates the `start` and `stop` indices of the corresponding local array.
- `comm_padding` integer, greater than or equal to zero.
- `boundary_padding` integer, greater than or equal to zero.

These integer values indicate the communication or boundary padding, respectively, for the local arrays. Currently only a single value for both `boundary_padding` and `comm_padding` is allowed for the entire dimension.

Cyclic

- `dist_type` is `'c'`
- `proc_grid_size` integer, greater than or equal to one.

The size of the process grid in this dimension. Equivalent to the number of local arrays in this dimension and determines the number of array sections.

- `size` integer, greater than or equal to zero.

The global size of the array in this dimension.

- `block_size` integer, optional. Greater than or equal to one.

If not present, equivalent to being present with value of one.

Unstructured

- `dist_type` is `'u'`
- `indices` sequence of one-dimensional numpy integer arrays or buffers.

  The `len(indices)` is the number of local unstructured arrays in this dimension.

  To compute the global size of the array in this dimension, compute `sum(len(ii) for ii in indices)`.

Not-distributed

The `'n'` distribution type is a convenience to specify that an array is not distributed along this dimension.

- `dist_type` is `'n'`
- `size` integer, greater than or equal to zero.

The global size of the array in this dimension.

---

classmethod `from_maps`(*context*, *maps*, *targets=None*)

Create a Distribution from a sequence of *Map*s.

Parameters

- **`context`** (*Context object*) –
- **`maps`** (*Sequence of Map objects*) –
- **`targets`** (*Sequence of int, optional*) – Sequence of engine target numbers. Default: all available

Returns

Return type *Distribution*

`get_dim_data_per_rank`()

`get_redist_plan`(*other_dist*)

`has_precise_index`

Does the client-side Distribution know precisely who owns all indices?

This can be used to determine whether one needs to use the *checked* version of *__getitem__* or *__setitem__* on LocalArrays.

`is_compatible`(*o*)

`localshapes`()

owning_ranks(*idxs*)
Returns a list of ranks that may *possibly* own the location in the *idxs* tuple.

For many distribution types, the owning rank is precisely known; for others, it is only probably known. When the rank is precisely known, *owning_ranks()* returns a list of exactly one rank. Otherwise, returns a list of more than one rank.

If the *idxs* tuple is out of bounds, raises *IndexError*.

owning_targets(*idxs*)
Like *owning_ranks()* but returns a list of targets rather than ranks.

Convenience method meant for IPython parallel usage.

reduce(*axes*)
Returns a new Distribution reduced along *axis*, i.e., the new distribution has one fewer dimension than *self*.

slice(*index_tuple*)
Make a new Distribution from a slice.

view(*new_dimsize=None*)
Generate a new Distribution for use with DistArray.view.

class distarray.globalapi.maps.**MapBase**
Bases: object

Base class for one-dimensional client-side maps.

Maps keep track of the relevant distribution information for a single dimension of a distributed array. Maps allow distributed arrays to keep track of which process to talk to when indexing and slicing.

Classes that inherit from *MapBase* must implement the *index_owners()* abstractmethod.

classmethod from_axis_dim_dicts(*axis_dim_dicts*)
Make a Map from a sequence of process-local dimension dictionaries.

There should be one such dimension dictionary per process.

classmethod from_global_dim_dict(*glb_dim_dict*)
Make a Map from a global dimension dictionary.

get_dimdicts()
Return a dim_dict per process in this dimension.

index_owners(*idx*)
Returns a list of process IDs in this dimension that might possibly own *idx*.

Raises *IndexError* if *idx* is out of bounds.

is_compatible(*map*)

class distarray.globalapi.maps.**NoDistMap**(*size*, *grid_size*)
Bases: *distarray.globalapi.maps.MapBase*

dist = 'n'

classmethod from_axis_dim_dicts(*axis_dim_dicts*)

classmethod from_global_dim_dict(*glb_dim_dict*)

get_dimdicts()

index_owners(*idx*)

is_compatible(*other*)

slice(*idx*)
Make a new Map from a slice.

slice_owners(*idx*)

view(*new_dimsize*)
Scale this map for the *view* method.

class distarray.globalapi.maps.**UnstructuredMap**(*size*, *grid_size*, *indices=None*)
Bases: *distarray.globalapi.maps.MapBase*

dist = 'u'

classmethod from_axis_dim_dicts(*axis_dim_dicts*)

classmethod from_global_dim_dict(*glb_dim_dict*)

get_dimdicts()

index_owners(*idx*)

distarray.globalapi.maps.**asdistribution**(*context*, *shape_or_dist*, *dist=None*, *grid_shape=None*, *targets=None*)

distarray.globalapi.maps.**choose_map**(*dist_type*)
Choose a map class given one of the distribution types.

distarray.globalapi.maps.**global_flat_indices**(*dim_data*)
Return a list of tuples of indices into the flattened global array.

Parameters **dim_data** (*dimension dictionary.*) –

Returns Each tuple is a (start, stop) interval into the flattened global array. All selected ranges comprise the indices for this dim_data's sub-array.

Return type list of 2-tuples of ints.

distarray.globalapi.maps.**map_from_global_dim_dict**(*global_dim_dict*)
Given a global_dim_dict return map.

distarray.globalapi.maps.**map_from_sizes**(*size*, *dist_type*, *grid_size*)
Returns an instance of the appropriate subclass of MapBase.

random Module

Contains the *Random* class that emulates *numpy.random* for *DistArray*.

class distarray.globalapi.random.**Random**(*context*)
Bases: object

normal(*shape_or_dist*, *loc=0.0*, *scale=1.0*)
Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [1], is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

Parameters

[1] P. R. Peebles Jr., "Central Limit Theorem" in "Probability, Random Variables and Random Signal Principles", 4th ed., 2001, pp. 51, 51, 125.

- **loc** (*float*) – Mean ("centre") of the distribution.
- **scale** (*float*) – Standard deviation (spread or "width") of the distribution.
- **shape_or_dist** (*shape tuple or Distribution object*) –

Notes

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2, is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

References

rand(*shape_or_dist*)
Random values over a given distribution.

Create a distarray of the given shape and propagate it with random samples from a uniform distribution over `[0, 1)`.

Parameters **shape_or_dist** (*shape tuple or Distribution object*) –

Returns **out** – Random values.

Return type *DistArray*

randint(*shape_or_dist*, *low*, *high=None*)
Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

Parameters

- **shape_or_dist** (*shape tuple or Distribution object*) –
- **low** (*int*) – Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).
- **high** (*int, optional*) – if provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

Returns **out** – DistArray of random integers from the appropriate distribution.

Return type DistArray of ints

randn(*shape_or_dist*)
Return samples from the "standard normal" distribution.

Parameters **shape_or_dist** (*shape tuple or Distribution object*) –

Returns **out** – A DistArray of floating-point samples from the standard normal distribution.

Return type *DistArray*

seed(*seed=None*)
Seed the random number generators on each engine.

Parameters **seed** (*None, int, or array of integers*) – Base random number seed to use on each engine. If None, then a non-deterministic seed is obtained from the operating system. Otherwise, the seed is used as passed, and the sequence of random numbers will be deterministic.

Each individual engine has its state adjusted so that it is different from each other engine. Thus, each engine will compute a different sequence of random numbers.

localapi Package

localapi Package

Modules dealing with the local index-space view of *DistArray*s.

In other words, the view from an engine.

construct Module

distarray.localapi.construct.**init_base_comm**(*comm*)
Sanitize an MPI.comm instance or create one.

distarray.localapi.construct.**init_comm**(*base_comm*, *grid_shape*)
Create an MPI communicator with a cartesian topology.

error Module

exception distarray.localapi.error.**IncompatibleArrayError**
Bases: *distarray.error.DistArrayError*

Exception class when arrays are incompatible.

exception distarray.localapi.error.**InvalidBaseCommError**
Bases: *distarray.error.DistArrayError*

Exception class when an object expected to be an MPI.Comm object is not one.

exception distarray.localapi.error.**InvalidDimensionError**
Bases: *distarray.error.DistArrayError*

Exception class when a specified dimension is invalid.

exception distarray.localapi.error.**NullCommError**
Bases: *distarray.error.DistArrayError*

Exception class when an MPI communicator is NULL.

format Module

Define a simple format for saving LocalArrays to disk with full information about them. This format, `.dnpy`, draws heavily from the `.npy` format specification from NumPy and from the data structure defined in the Distributed Array Protocol.

Version numbering The version numbering of this format is independent of DistArray's and the Distributed Array Protocol's version numberings.

Format Version 1.0 The first 6 bytes are a magic string: exactly `\x93DARRY`.

The next 1 byte is an unsigned byte: the major version number of the file format, e.g. `\x01`.

The next 1 byte is an unsigned byte: the minor version number of the file format, e.g. `\x00`. Note: the version of the file format is not tied to the version of the DistArray package.

The next 2 bytes form a little-endian unsigned short int: the length of the header data HEADER_LEN.

The next HEADER_LEN bytes form the header data describing the distribution of this chunk of the LocalArray. It is an ASCII string which contains a Python literal expression of a dictionary. It is terminated by a newline (`\n`) and padded with spaces (`\x20`) to make the total length of `magic string + 4 + HEADER_LEN` be evenly divisible by 16 for alignment purposes.

The dictionary contains two keys, both described in the Distributed Array Protocol:

"__version__" [str] Version of the Distributed Array Protocol used in this header.

"dim_data" [tuple of dict] One dictionary per array dimension; see the Distributed Array Protocol for the details of this data structure.

For repeatability and readability, the dictionary keys are sorted in alphabetic order. This is for convenience only. A writer SHOULD implement this if possible. A reader MUST NOT depend on this.

Following this header is the output of `numpy.save` for the underlying data buffer. This contains the full output of `save`, beginning with the magic number for `.npy` files, followed by the `.npy` header and array data.

Notes

The `.npy` format, including reasons for creating it and a comparison of alternatives, is described fully in the "npy-format" NEP and in the module docstring for `numpy.lib.format`.

`distarray.localapi.format.`**`magic`**(*major*, *minor*, *prefix=<MagicMock name='mock.asbytes()' id='140675672494672'>*)

Return the magic string for the given file format version.

Parameters

- **`major`** (*int in [0, 255]*) –
- **`minor`** (*int in [0, 255]*) –
- **`prefix`** (*bytes*) – The magic prefix to concatenate with version number

Returns **magic**

Return type bytes

Raises `ValueError` – if the version cannot be formatted.

`distarray.localapi.format.`**`read_localarray`**(*fp*)

Read a LocalArray from an .dnpy file.

Parameters **`fp`** (*file_like object*) – If this is not a real file object, then this may take extra memory and time.

Returns **distbuffer** – The Distributed Array Protocol structure created from the data on disk.

Return type dict

Raises `ValueError` – If the data is invalid.

distarray.localapi.format.**read_localarray_header**(*fp*, *version*)
Read an array header from a filelike object using the 1.0 file format version.

This will leave the file object located just after the header.

Parameters

- **fp** (*filelike object*) – A file object or something with a *.read()* method like a file.
- **version** (*tuple of int*) –

Returns

- **__version__** (*str*) – Version of the Distributed Array Protocol used.
- **dim_data** (*tuple*) – A tuple containing a dictionary for each dimension of the underlying array, as described in the Distributed Array Protocol.

Raises ValueError – If the data is invalid.

distarray.localapi.format.**read_magic**(*fp*, *prefix=<MagicMock name='mock.asbytes()' id='140675672494672'>*, *prefix_len=2*)
Read the magic string to get the version of the file format.

Parameters

- **fp** (*filelike object*) –
- **prefix** (*bytes*) – Magic prefix to look for
- **prefix_len** (*int*) – Number of bytes in *prefix*

Returns

- **major** (*int*)
- **minor** (*int*)

distarray.localapi.format.**write_localarray**(*fp*, *larr*, *version=(1, 0)*)
Write a LocalArray to a .dnpy file, including a header.

The `__version__` and `dim_data` keys from the Distributed Array Protocol are written to a header, then `numpy.save` is used to write the value of the `buffer` key.

Parameters

- **fp** (*file_like object*) – An open, writable file object, or similar object with a `.write()` method.
- **larr** (LocalArray) – The array to write to disk.
- **version** (*(int, int), optional*) – The version number of the file format. Default: (1, 0)

Raises

- ValueError – If the array cannot be persisted.
- **Various other errors** – If the underlying numpy array contains Python objects as part of its dtype, the process of pickling them may raise various errors if the objects are not picklable.

distarray.localapi.format.**write_localarray_header**(*fp*, *d*, *version=None*)
Write the header for a localarray and return the version used

Parameters

- **fp** (*filelike object*) –
- **d** (*dict*) – This has the appropriate entries for writing its string representation to the header of the file.

- **version** (*tuple or None*) – None means use oldest that works explicit version will raise a ValueError if the format does not allow saving this data. Default: None

Returns **version** – the file version which needs to be used to store the data

Return type tuple of int

localarray Module

The *LocalArray* data structure.

DistArray objects are proxies for collections of *LocalArray* objects (that usually reside on engines).

class distarray.localapi.localarray.**GlobalIndex**(*distribution*, *ndarray*)
Bases: object

Object which provides access to global indexing on LocalArrays.

checked_getitem(*global_inds*)

checked_setitem(*global_inds*, *value*)

get_slice(*global_inds*, *new_distribution*)

class distarray.localapi.localarray.**GlobalIterator**(*arr*)
Bases: distarray.externals.six.Iterator

class distarray.localapi.localarray.**LocalArray**(*distribution*, *dtype=None*, *buf=None*)
Bases: object

Distributed memory Python arrays.

__array_wrap__(*obj*, *context=None*)
Return a LocalArray based on obj.

This method constructs a new LocalArray object using the distribution from self and the buffer from obj.

This is used to construct return arrays for ufuncs.

__distarray__()
Returns the data structure required by the DAP.

DAP = Distributed Array Protocol

See the project's documentation for the Protocol's specification.

__getitem__(*index*)
Get a local item.

__setitem__(*index*, *value*)
Set a local item.

asdist_like(*other*)
Return a version of self that has shape, dist and grid_shape like *other*.

astype(*newdtype*)
Return a copy of this LocalArray with a new underlying dtype.

cart_coords

comm

comm_rank

comm_size

compatibility_hash()

coords_from_rank(*rank*)

copy()
Return a copy of this LocalArray.

dim_data

dist

dtype

fill(*scalar*)

classmethod from_distarray(*comm*, *obj*)
Make a LocalArray from Distributed Array Protocol data structure.

An object that supports the Distributed Array Protocol will have a *__distarray__* method that returns the data structure described here:

https://github.com/enthought/distributed-array-protocol

Parameters **obj** (an object with a *__distarray__* method or a dict) – If a dict, it must conform to the structure defined by the distributed array protocol.

Returns A LocalArray encapsulating the buffer of the original data. No copy is made.

Return type *LocalArray*

global_from_local(*local_ind*)

global_limits(*dim*)

global_shape

global_size

grid_shape

itemsize

local_data

local_from_global(*global_ind*)

local_shape

local_size

local_view(*dtype=None*)

nbytes

ndarray

ndim

pack_index(*inds*)

rank_from_coords(*coords*)

sync()

unpack_index(*packed_ind*)

view(*distribution*, *dtype*)
Return a new LocalArray whose underlying *ndarray* is a view on *self.ndarray*.

class distarray.localapi.localarray.**LocalArrayBinaryOperation**(*numpy_ufunc*)
Bases: object

class distarray.localapi.localarray.**LocalArrayUnaryOperation**(*numpy_ufunc*)
Bases: object

distarray.localapi.localarray.**arecompatible**(*a*, *b*)
Do these arrays have the same compatibility hash?

distarray.localapi.localarray.**compact_indices**(*dim_data*)
Given a *dim_data* structure, return a tuple of compact indices.

For every dimension in *dim_data*, return a representation of the indices indicated by that dim_dict; return a slice if possible, else, return the list of global indices.

Parameters **dim_data** (*tuple of dict*) – A dict for each dimension, with the data described here: https://github.com/enthought/distributed-array-protocol we use only the indexing related keys from this structure here.

Returns **index** – Efficient structure usable for indexing into a numpy-array-like data structure.

Return type tuple of slices and/or lists of int

distarray.localapi.localarray.**empty**(*distribution*, *dtype=<type 'float'>*)
Create an empty LocalArray.

distarray.localapi.localarray.**empty_like**(*arr*, *dtype=None*)
Create an empty LocalArray with a distribution like *arr*.

distarray.localapi.localarray.**fromfunction**(*function*, *distribution*, ***kwargs*)

distarray.localapi.localarray.**fromndarray_like**(*ndarray*, *like_arr*)
Create a new LocalArray like *like_arr* with buffer set to *ndarray*.

distarray.localapi.localarray.**get_printoptions**()

distarray.localapi.localarray.**load_dnpy**(*comm*, *file*)
Load a LocalArray from a `.dnpy` file.

Parameters **file** (*file-like object or str*) – The file to read. It must support `seek()` and `read()` methods.

Returns **result** – A LocalArray encapsulating the data loaded.

Return type *LocalArray*

distarray.localapi.localarray.**load_hdf5**(*comm*, *filename*, *dim_data*, *key='buffer'*)
Load a LocalArray from an `.hdf5` file.

Parameters

- **filename** (*str*) – The filename to read.
- **dim_data** (*tuple of dict*) – A dict for each dimension, with the data described here: https://github.com/enthought/distributed-array-protocol, describing which portions of the HDF5 file to load into this LocalArray, and with what metadata.
- **comm** (*MPI comm object*) –
- **key** (*str, optional*) – The identifier for the group to load the LocalArray from (the default is 'buffer').

Returns **result** – A LocalArray encapsulating the data loaded.

Return type *LocalArray*

Note: For *dim_data* dimension dictionaries containing unstructured ('u') distribution types, the indices selected by the *'indices'* key must be in increasing order. This is a limitation of h5py / hdf5.

distarray.localapi.localarray.**load_npy**(*comm*, *filename*, *dim_data*)
Load a LocalArray from a `.npy` file.

Parameters

- **filename** (*str*) – The file to read.
- **dim_data** (*tuple of dict*) – A dict for each dimension, with the data described here: https://github.com/enthought/distributed-array-protocol, describing which portions of the HDF5 file to load into this LocalArray, and with what metadata.
- **comm** (*MPI comm object*) –

Returns **result** – A LocalArray encapsulating the data loaded.

Return type *LocalArray*

distarray.localapi.localarray.**local_reduction**(*out_comm*, *reducer*, *larr*, *ddpr*, *dtype*, *axes*)
Entry point for reductions on local arrays.

Parameters

- **reducer** (*callable*) – Performs the core reduction operation.
- **out_comm** (*MPI Comm instance.*) – The MPI communicator for the result of the reduction. Is equal to MPI.COMM_NULL when this rank is not part of the output communicator.
- **larr** (LocalArray) – Input. Defined for all ranks.
- **ddpr** (*sequence of dim-data dictionaries.*) –
- **axes** (*Sequence of ints or None.*) –

Returns When out_comm == MPI.COMM_NULL, returns None. Otherwise, returns the LocalArray section of the reduction result.

Return type LocalArray or None

distarray.localapi.localarray.**make_local_slices**(*local_arr*, *glb_indices*)

distarray.localapi.localarray.**max_reducer**(*reduce_comm*, *larr*, *out*, *axes*, *dtype*)
Core reduction function for max.

distarray.localapi.localarray.**mean_reducer**(*reduce_comm*, *larr*, *out*, *axes*, *dtype*)
Core reduction function for mean.

distarray.localapi.localarray.**min_reducer**(*reduce_comm*, *larr*, *out*, *axes*, *dtype*)
Core reduction function for min.

distarray.localapi.localarray.**mpi_print**(**args*, ***kwargs*)

distarray.localapi.localarray.**ndenumerate**(*arr*)

distarray.localapi.localarray.**ones**(*distribution*, *dtype=<type 'float'>*)
Create a LocalArray filled with ones.

distarray.localapi.localarray.**redistribute**(*comm*, *plan*, *la_from*, *la_to*)

distarray.localapi.localarray.**redistribute_general**(*comm*, *plan*, *la_from*, *la_to*)

distarray.localapi.localarray.**save_dnpy**(*file*, *arr*)
Save a LocalArray to a `.dnpy` file.

Parameters

- **file** (*file-like object or str*) – The file or filename to which the data is to be saved.
- **arr** (LocalArray) – Array to save to a file.

distarray.localapi.localarray.**save_hdf5**(*filename*, *arr*, *key='buffer'*, *mode='a'*)
Save a LocalArray to a dataset in an `.hdf5` file.

Parameters

- **filename** (*str*) – Name of file to write to.
- **arr** (LocalArray) – Array to save to a file.
- **key** (*str, optional*) – The identifier for the group to save the LocalArray to (the default is 'buffer').
- **mode** (*optional, {'w', 'w-', 'a'}, default 'a'*) –

  'w' Create file, truncate if exists

  'w-' Create file, fail if exists

  'a' Read/write if exists, create otherwise (default)

distarray.localapi.localarray.**set_printoptions**(*precision=None*, *threshold=None*, *edgeitems=None*, *linewidth=None*, *suppress=None*)

distarray.localapi.localarray.**std_reducer**(*reduce_comm*, *larr*, *out*, *axes*, *dtype*)
Core reduction function for std.

distarray.localapi.localarray.**sum_reducer**(*reduce_comm*, *larr*, *out*, *axes*, *dtype*)
Core reduction function for sum.

distarray.localapi.localarray.**var_reducer**(*reduce_comm*, *larr*, *out*, *axes*, *dtype*)
Core reduction function for var.

distarray.localapi.localarray.**zeros**(*distribution*, *dtype=<type 'float'>*)
Create a LocalArray filled with zeros.

distarray.localapi.localarray.**zeros_like**(*arr*, *dtype=<type 'float'>*)
Create a LocalArray of zeros with a distribution like *arr*.

maps Module

Classes to manage the distribution-specific aspects of a LocalArray.

The Distribution class is the main entry point and is meant to be used by LocalArrays to help translate between local and global index spaces. It manages *ndim* one-dimensional map objects.

The one-dimensional map classes BlockMap, CyclicMap, BlockCyclicMap, and UnstructuredMap all manage the mapping tasks for their particular dimension. All are subclasses of MapBase. The reason for the several subclasses is to allow more compact and efficient operations.

class distarray.localapi.maps.**BlockCyclicMap**(*global_size*, *grid_size*, *grid_rank*, *start*, *block_size*)
Bases: *distarray.localapi.maps.MapBase*

One-dimensional block cyclic map class.

dim_dict

dist = 'c'

global_from_local_index(*lidx*)

global_iter

global_slice
Return a slice representing the global index space of this dimension; only possible for block_size == 1.

local_from_global_index(*gidx*)

size

class distarray.localapi.maps.**BlockMap**(*global_size*, *grid_size*, *grid_rank*, *start*, *stop*)
Bases: *distarray.localapi.maps.MapBase*

One-dimensional block map class.

dim_dict

dist = 'b'

global_from_local_index(*lidx*)

global_from_local_slice(*lidx*)

global_iter

global_slice
Return a slice representing the global index space of this dimension.

local_from_global_index(*gidx*)

local_from_global_slice(*gidx*)

size

class distarray.localapi.maps.**CyclicMap**(*global_size*, *grid_size*, *grid_rank*, *start*)
Bases: *distarray.localapi.maps.MapBase*

One-dimensional cyclic map class.

dim_dict

dist = 'c'

global_from_local_index(*lidx*)

global_iter

global_slice
Return a slice representing the global index space of this dimension.

local_from_global_index(*gidx*)

size

class distarray.localapi.maps.**Distribution**(*comm*, *dim_data*)
Bases: object

Multi-dimensional Map class.

Manages one or more one-dimensional map classes.

cart_coords

comm_rank

comm_size

coords_from_rank(*rank*)

dim_data

dist

classmethod from_shape(*comm*, *shape*, *dist=None*, *grid_shape=None*)
Create a Distribution from a *shape* and optional arguments.

global_from_local(*local_ind*)
Given *local_ind* indices, translate into global indices.

global_shape

global_size

global_slice
Return a slice representing the global index space of this dimension.

grid_shape

local_flat_from_local(*local_ind*)

local_from_global(*global_ind*)
Given *global_ind* indices, translate into local indices.

local_shape

local_size

ndim

rank_from_coords(*coords*)

class distarray.localapi.maps.**MapBase**
Bases: object

Base class for all one dimensional Map classes.

class distarray.localapi.maps.**UnstructuredMap**(*global_size*, *grid_size*, *grid_rank*, *indices*)
Bases: *distarray.localapi.maps.MapBase*

One-dimensional unstructured map class.

dim_dict

dist = 'u'

global_from_local_index(*lidx*)

global_iter

local_from_global_index(*gidx*)

size

distarray.localapi.maps.**map_from_dim_dict**(*dd*)
Factory function that returns a 1D map for a given dimension dictionary.

mpiutils Module

Entry point for MPI.

distarray.localapi.mpiutils.**create_comm_of_size**(*size=4*)
Create a subcommunicator of COMM_PRIVATE of given size.

distarray.localapi.mpiutils.**create_comm_with_list**(*nodes*, *base_comm=None*)
Create a subcommunicator of base_comm with a list of ranks.

If base_comm is not specified, defaults to COMM_PRIVATE.

distarray.localapi.mpiutils.**get_base_comm**()

distarray.localapi.mpiutils.**get_comm_private**()

distarray.localapi.mpiutils.**mpi_type_for_ndarray**(*a*)

distarray.localapi.mpiutils.**set_base_comm**(*comm*)

proxyize Module

class distarray.localapi.proxyize.**Proxy**(*name*, *obj*, *module_name*)
Bases: object

cleanup()

dereference()
Callable only on the engines.

class distarray.localapi.proxyize.**Proxyize**
Bases: object

Callable that, given an object, returns a Proxy object.

You must call *set_state* on the instance before you can "call" it.

__call__(*obj*)
Return a *Proxy* object given an object *obj*.

next_name()

set_state(*state*)

str_counter()
Return the str value of *self.count*, then increment its value.

random Module

Pseudo-random number generation routines for local arrays.

This module provides a number of routines for generating random numbers, from a variety of probability distributions.

distarray.localapi.random.**beta**(*a*, *b*, *distribution=None*)
Return an array with random numbers from the beta probability distribution.

Parameters

- **a** (*float*) – Parameter that describes the beta probability distribution.
- **b** (*float*) – Parameter that describes the beta probability distribution.
- **distribution** (*The desired distribution of the array.*) – If None, then a normal NumPy array is returned. Otherwise, a LocalArray with this distribution is returned.

Returns

Return type An array with random numbers.

distarray.localapi.random.**label_state**(*comm*)
Label/personalize the random generator state for the local rank.

This ensures that each separate engine, when using the same global seed, will generate a different sequence of pseudo-random numbers.

distarray.localapi.random.**normal**(*loc=0.0*, *scale=1.0*, *distribution=None*)
Return an array with random numbers from a normal (Gaussian) probability distribution.

Parameters

- **loc** (*float*) – The mean (or center) of the probability distribution.
- **scale** (*float*) – The standard deviation (or width) of the probability distribution.
- **distribution** (*The desired distribution of the array.*) – If None, then a normal NumPy array is returned. Otherwise, a LocalArray with this distribution is returned.

Returns

Return type An array with random numbers.

distarray.localapi.random.**rand**(*distribution=None*)
Return an array with random numbers distributed over the interval [0, 1).

Parameters **distribution** (*The desired distribution of the array.*) – If None, then a normal NumPy array is returned. Otherwise, a LocalArray with this distribution is returned.

Returns

Return type An array with random numbers.

distarray.localapi.random.**randint**(*low*, *high=None*, *distribution=None*)
Return random integers from low (inclusive) to high (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

Parameters

- **low** (*int*) – Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the highest such integer).
- **high** (*int, optional*) – If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).
- **distribution** (*The desired distribution of the array.*) – If None, then a normal NumPy array is returned. Otherwise, a LocalArray with this distribution is returned.

Returns

Return type An array with random numbers.

distarray.localapi.random.**randn**(*distribution=None*)
Return a sample (or samples) from the "standard normal" distribution.

Parameters **distribution** (*The desired distribution of the array.*) – If None, then a normal NumPy array is returned. Otherwise, a LocalArray with this distribution is returned.

Returns

Return type An array with random numbers.

plotting Package

plotting Package

Plotting functions for distarrays.

plotting Module

Plotting functions for distarrays.

distarray.plotting.plotting.**cmap_discretize**(*cmap*, *N*)

Create a discrete colormap from the continuous colormap cmap.

Parameters

- **cmap** (*colormap instance, or string*) – The continuous colormap, as object or name, to make discrete. For example, matplotlib.cm.jet, or 'jet'.
- **N** (*int*) – The number of discrete colors desired.

Returns The desired discrete colormap.

Return type colormap

Example

```
>>> x = resize(arange(100), (5,100))
>>> djet = cmap_discretize(cm.jet, 5)
>>> pyplot.imshow(x, cmap=djet)
```

distarray.plotting.plotting.**create_discrete_colormaps**(*num_values*)

Create colormap objects for a discrete colormap.

Parameters **num_values** (*The number of distinct colors to use.*) –

Returns **cmap, norm, text_colors** – The matplotlib colormap, norm, and recommended text colors. text_colors is an array of length num_values, with each entry being a nice color for text drawn on top of the colormap selection.

Return type tuple

distarray.plotting.plotting.**get_ranks**(*arr*)

Given a distarray arr, return a distarray with the same shape, but with the elements equal to the rank of the process the element is on.

distarray.plotting.plotting.**plot_array_distribution**(*darray*, *process_coords*, *title=None*, *xlabel=None*, *ylabel=None*, *yflip=False*, *cell_label=True*, *legend=False*, *global_plot_filename=None*, *local_plot_filename=None*, **args*, ***kwargs*)

Plot a distarray's memory layout. It can be 1D or 2D. Elements are colored according to the process they are on.

Parameters

- **darray** (DistArray) – The distributed array to plot.
- **process_coords** (*List of tuples.*) – The process grid coordinates.

- **title** (*string*) – Text label for the plot title, or None.
- **xlabel** (*string*) – Text label for the x-axis, or None.
- **ylabel** (*string*) – Text label for the y-axis, or None.
- **yflip** (*bool*) – If True, then the y-axis increases downwards, to match the layout when printing the array itself.
- **cell_label** (*bool*) – If True, then each cell in the plot is labeled with the array value. This can look cluttered for large arrays.
- **legend** (*bool*) – If True, then a colorbar legend is drawn to label the colors.
- **global_plot_filename** (*string*) – Output filename for the global array plot image.
- **local_plot_filename** (*string*) – Output filename for the local array plot image.

Returns The process assignment array, as a DistArray.

Return type out

distarray.plotting.plotting.**plot_local_array_subfigure**(*subfig*, *local_array*, *process*, *coord*, *colormap_objects*, **args*, ***kwargs*)

Plot a single local_array into a matplotlib subfigure.

distarray.plotting.plotting.**plot_local_arrays**(*darray*, *process_coords*, *colormap_objects*, *filename*)

Plot the local arrays as a multi-figure matplotlib plot.

9.2 Building HDF5 and h5py for DistArray

If you want to use DistArray's parallel IO capabilities on HDF5 files, parallel-enabled HDF5 and h5py installations are required. Unfortunately, installing these can be somewhat of a pain.

9.2.1 Update 2015-10-05

The "Original Notes" below are pretty old at this point. Recently I have had success installing a parallel-enabled HDF5 (1.8.15) using Homebrew on OS X:

```
$ brew install hdf5 --with-mpi
```

The instructions for building `h5py` on top of a parallel hdf5 have also changed, but are available here.

9.2.2 Original Notes

These are notes from trying to build HDF5 1.8.12 and h5py 2.2.1 against mpi4py 1.3 and openmpi-1.6.5 on OS X 10.8.5.

9.2.3 HDF5

Download the HDF5 source (1.8.12) and configure it with parallel support. From the source directory:

```
$ CFLAGS=-O0 CC=/Users/robertgrant/localroot/bin/mpicc ./configure --enable-shared --enable-parallel
```

The CFLAGS setting is to get around a known problem with the tests on OS X 10.8 (http://www.hdfgroup.org/HDF5/release/known_problems/).

Build it:

```
$ make
```

Test it:

```
$ make check
```

This produced some errors related to ph5diff, which the website claims are "not valid errors", so I ignored them (http://www.hdfgroup.org/HDF5/faq/parallel.html#ph5difftest).

Install HDF5:

```
$ make install
```

9.2.4 h5py

Build h5py against this version of HDF5. Without setting `HDF5_DIR`, on my system the build found Canopy's serial version of HDF5. In the h5py source directory:

```
$ HDF5_DIR=/Users/robertgrant/localroot/ CC=mpicc python setup.py build --mpi
```

This gives me an error about "MPI Message" addressed here:

```
https://github.com/h5py/h5py/issues/401
```

After patching api_compat.h as suggested, it builds. One could also use the `master` version of h5py from GitHub instead of the latest release.

Run the tests:

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

and install h5py:

```
$ python setup.py install
```

You should now be able to run the example code listed here:

```
http://docs.h5py.org/en/latest/mpi.html#using-parallel-hdf5-from-h5py
```

9.3 Notes on building environment-modules

9.3.1 Update 2015-10-05

Below are Mark Kness' notes on building and using `environment-modules`. I have since found lmod to be a slightly better documented and supported alternative.

- Bob Grant

9.3.2 Environment Modules

environment-modules is a tool, written with Tcl, that makes it convenient to switch environment settings. It is not required to use DistArray, but we find it useful in development. It is a difficult name to google. I had to build it from source, and made some notes of my steps, which will hopefully be helpful for others that build this.

There seems to be some version available to `apt-get` for Debian. But I read suggestions not to mix Debian and Ubuntu packages, and as I have Ubuntu, I did not try and configure my `apt-get` to look at the Debian packages. So I installed from source, with notes as follows.

These specific notes are from an installation from source for Linux Mint (Ubuntu), done by Mark Kness. These actions were based on the INSTALL document in the modules source and the Geoghegan link.

```
$ sudo apt-get install tcl tcl8.4-dev
```

This seemed to run ok.

```
$ tar xvvf modules-3.2.10.tar.gz
```

I had already downloaded this. Double v means extra verbose.

```
$ cd modules-3.2.10
$ gedit README
$ gedit INSTALL
$ gedit INSTALL.RH7x
```

Read the installation notes!

```
$ ./configure
```

First step is to run this and see how far it gets. Tcl is the likely problem here.

I got the following messages from ./configure...:

```
checking for Tcl configuration (tclConfig.sh)... found /usr/lib/tcl8.4/tclConfig.sh
checking for existence of tclConfig.sh... loading
checking for Tcl version... 8.5
checking TCL_VERSION... 8.5
checking TCL_LIB_SPEC... -L/usr/lib -ltcl8.4
checking TCL_INCLUDE_SPEC... -I/usr/include/tcl8.4
checking for TclX configuration (tclxConfig.sh)... not found
checking for TclX version... using 8.5
checking TCLX_VERSION... 8.5
checking TCLX_LIB_SPEC... TCLX_LIB_SPEC not found, need to use --with-tclx-lib
checking TCLX_INCLUDE_SPEC... TCLX_INCLUDE_SPEC not found, need to use --with-tclx-inc
configure: WARNING: will use MODULEPATH=/usr/local/Modules/modulefiles : rerun configure using --with
configure: WARNING: will use VERSIONPATH=/usr/local/Modules/versions : rerun configure using --with-v
```

It seems that TCL_VERSION, TCL_LIB_SPEC, and TCL_INCLUDE_SPEC were all found ok. (The TCLX variants are not found but that is different and not a problem.) Generally it seems like Tcl is ok, except perhaps for some 8.4 vs 8.5 version inconsistency. A non-default path for the module files themselves seems recommended, so...

```
$ cd ~
$ mkdir modules
```

This created `/home/mkness/modules` on my machine. The install notes suggest that one make a non-default location for these. This directory name was an arbitrary choice.

```
$ cd modules-3.2.10
$ ./configure --with-module-path=~/modules
```

Seemed ok. I ignored the version and prefix path options.

```
$ make
```

Seemed basically ok, a few warnings.

```
$ ./modulecmd sh
```

I got the usage instructions, and NOT any Tcl messages. Ok!

```
$ sudo make install
```

Seemed to run ok. Got permission errors without sudo.

```
$ cd /usr/local/Modules
$ sudo ln -s 3.2.10 default
```

Setup symbolic link named 'default' to point to the installed version.

```
$ cd ~
$ /usr/local/Modules/default/bin/add.modules
```

This script is supposed to update my local .bashrc and similar files to have access to the Modules stuff. For me, it modified .bashrc and .profile. But if I say 'module', I get an error about an invalid path. It seems that MODULE_VERSION is not defined, so I added `export MODULE_VERSION=default` to the top of my .bashrc.

At this point I can say 'module' at the command line and I get the usage instructions. But 'module avail' dislikes the lack of an environment variable MODULEPATH. So I also add `export MODULEPATH=~/modules` to my .bashrc. This path matches the –with-module-path argument to ./configure.

Now it works!

9.3.3 References

http://modules.sourceforge.net/ The main page for the modules package. It provides a source download: modules-3.2.10.tar.gz

http://sourceforge.net/p/modules/wiki/FAQ/ FAQ for the modules package.

http://nickgeoghegan.net/linux/installing-environment-modules Build instructions for environment-modules. I partially followed these but with several changes.

http://packages.debian.org/wheezy/environment-modules http://packages.debian.org/wheezy/amd64/environment-modules/download http://packages.debian.org/unstable/main/environment-modules Debian package for environment-modules. Note that this is two different places.

http://packages.debian.org/search?keywords=tcl&searchon=names&suite=stable&section=all Debian package for Tcl.

9.4 Licence for *six.py* version 1.5.2

CHAPTER 10

Release Notes

10.1 DistArray 0.2: development release

Documentation: http://distarray.readthedocs.org License: Three-clause BSD Python versions: 2.7 and 3.3 OS support: *nix and Mac OS X

DistArray aims to bring the strengths of NumPy to data-parallel high-performance computing. It provides distributed multi-dimensional NumPy-like arrays and distributed ufuncs, distributed IO capabilities, and can integrate with external distributed libraries, like Trilinos. DistArray works with NumPy and builds on top of it in a flexible and natural way.

Brian Granger started DistArray as a NASA-funded SBIR project in 2008. Enthought picked it up as part of a DOE Phase II SBIR [0] to provide a generally useful distributed array package. It builds on IPython, IPython.parallel, NumPy, MPI, and interfaces with the Trilinos suite of distributed HPC solvers (via PyTrilinos) [1].

Distarray:

- has a client-engine (or master-worker) process design – data resides on the worker processes, commands are initiated from master;
- allows full control over what is executed on the worker processes and integrates transparently with the master process;
- allows direct communication between workers bypassing the master process for scalability;
- integrates with IPython.parallel for interactive creation and exploration of distributed data;
- supports distributed ufuncs (currently without broadcasting);
- builds on and leverages MPI via MPI4Py in a transparent and user-friendly way;
- supports NumPy-like structured multidimensional arrays;
- has basic support for unstructured arrays;
- supports user-controllable array distributions across workers (block, cyclic, block-cyclic, and unstructured) on a per-axis basis;
- has a straightforward API to control how an array is distributed;
- has basic plotting support for visualization of array distributions;
- separates the array's distribution from the array's data – useful for slicing, reductions, redistribution, broadcasting, all of which will be implemented in coming releases;
- implements distributed random arrays;

- supports .npy-like flat-file IO and hdf5 parallel IO (via h5py); leverages MPI-based IO parallelism in an easy-to-use and transparent way; and
- supports the distributed array protocol [2], which allows independently developed parallel libraries to share distributed arrays without copying, analogous to the PEP-3118 new buffer protocol.
- This is the first public development release. DistArray is not ready for real-world use, but we want to get input from the larger scientific-Python community to help drive its development. The API is changing rapidly and we are adding many new features on a fast timescale. For that reason, DistArray is currently implemented in pure Python for maximal flexibility. Performance improvements are coming.

The 0.2 release's goals are to provide the components necessary to support upcoming features that are non-trivial to implement in a distributed environment.

Planned features for upcoming releases:

- Distributed reductions
- Distributed slicing
- Distributed broadcasting
- Distributed fancy indexing
- Re-distribution methods
- Integration with Trilinos [1] and other packages [3] that subscribe to the distributed array protocol [2]
- Lazy evaluation and deferred computation for latency hiding
- Out-of-core computations
- Extensive examples, tutorials, documentation
- Support for distributed sorting and other non-trivial distributed algorithms
- MPI-only communication for non-interactive deployment on clusters and supercomputers
- End-user control over communication and temporary array creation, and other performance aspects of distributed computations

[0] http://www.sbir.gov/sbirsearch/detail/410257 [1] http://trilinos.org/ [2] http://distributed-array-protocol.readthedocs.org/en/rel-0.10.0/ [3] http://www.mcs.anl.gov/petsc/

10.2 DistArray 0.3: development release

Documentation: http://distarray.readthedocs.org

License: Three-clause BSD

Python versions: 2.7 and 3.3

OS support: *nix and Mac OS X

10.2.1 What is DistArray?

DistArray aims to bring the strengths of NumPy to data-parallel high-performance computing. It provides distributed multi-dimensional NumPy-like arrays and distributed ufuncs, distributed IO capabilities, and can integrate with external distributed libraries, like Trilinos. DistArray works with NumPy and builds on top of it in a flexible and natural way.

10.2.2 0.3 Release

This is the second development release.

Noteworthy improvements in 0.3 include:

- support for distributions over a subset of processes;
- distributed reductions with a simple NumPy-like API: `da.sum(axis=3)`;
- an `apply()` function for easier computation with process-local data;
- performance improvements and reduced communication overhead;
- cleanup, renamings, and refactorings;
- test suite improvements for parallel testing; and
- start of a more frequent release schedule.

DistArray is not ready for real-world use. We want to get input from the larger scientific-Python community to help drive its development. The API is changing rapidly and we are adding many new features on a fast timescale. DistArray is currently implemented in pure Python for maximal flexibility. Performance improvements are ongoing.

10.2.3 Existing features

Distarray:

- has a client-engine (or master-worker) process design – data resides on the worker processes, commands are initiated from master;
- allows full control over what is executed on the worker processes and integrates transparently with the master process;
- allows direct communication between workers bypassing the master process for scalability;
- integrates with IPython.parallel for interactive creation and exploration of distributed data;
- supports distributed ufuncs (currently without broadcasting);
- builds on and leverages MPI via MPI4Py in a transparent and user-friendly way;
- supports NumPy-like structured multidimensional arrays;
- has basic support for unstructured arrays;
- supports user-controllable array distributions across workers (block, cyclic, block-cyclic, and unstructured) on a per-axis basis;
- has a straightforward API to control how an array is distributed;
- has basic plotting support for visualization of array distributions;
- separates the array's distribution from the array's data – useful for slicing, reductions, redistribution, broadcasting, and other operations;
- implements distributed random arrays;
- supports `.npy`-like flat-file IO and hdf5 parallel IO (via `h5py`); leverages MPI-based IO parallelism in an easy-to-use and transparent way; and
- supports the distributed array protocol *[protocol]*, which allows independently developed parallel libraries to share distributed arrays without copying, analogous to the PEP-3118 new buffer protocol.

10.2.4 Planned features and roadmap

- Distributed slicing
- Re-distribution methods
- Integration with Trilinos *[Trilinos]* and other packages *[petsc]* that subscribe to the distributed array protocol *[protocol]*
- Distributed broadcasting
- Distributed fancy indexing
- MPI-only communication for non-interactive deployment on clusters and supercomputers
- Lazy evaluation and deferred computation for latency hiding
- Out-of-core computations
- Extensive examples, tutorials, documentation
- Support for distributed sorting and other non-trivial distributed algorithms
- End-user control over communication and temporary array creation, and other performance aspects of distributed computations

10.2.5 History

Brian Granger started DistArray as a NASA-funded SBIR project in 2008. Enthought picked it up as part of a DOE Phase II SBIR *[SBIR]* to provide a generally useful distributed array package. It builds on IPython, IPython.parallel, NumPy, MPI, and interfaces with the Trilinos suite of distributed HPC solvers (via PyTrilinos *[Trilinos]*).

10.3 DistArray 0.4 development release

Documentation: http://distarray.readthedocs.org

License: Three-clause BSD

Python versions: 2.7 and 3.3

OS support: *nix and Mac OS X

10.3.1 What is DistArray?

DistArray aims to bring the strengths of NumPy to data-parallel high-performance computing. It provides distributed multi-dimensional NumPy-like arrays and distributed ufuncs, distributed IO capabilities, and can integrate with external distributed libraries like Trilinos. DistArray works with NumPy and builds on top of it in a flexible and natural way.

10.3.2 0.4 Release

This is the third development release.

Noteworthy improvements in 0.4 include:

- basic slicing support;
- significant performance enhancements;

- reduction methods now support boolean arrays;
- an IPython notebook that demos basic functionality; and
- many bug fixes, API improvements, and refactorings.

DistArray is nearly ready for real-world use. The project is evolving rapidly and input from the larger scientific-Python community is very valuable and helps drive development.

10.3.3 Existing features

DistArray:

- has a client-engine (or master-worker) process design – data resides on the worker processes, and commands are initiated from master;
- allows full control over what is executed on the worker processes and integrates transparently with the master process;
- allows direct communication between workers, bypassing the master process for scalability;
- integrates with IPython.parallel for interactive creation and exploration of distributed data;
- supports distributed ufuncs (currently without broadcasting);
- builds on and leverages MPI via MPI4Py in a transparent and user-friendly way;
- supports NumPy-like multidimensional arrays;
- has basic support for unstructured arrays;
- supports user-controllable array distributions across workers (block, cyclic, block-cyclic, and unstructured) on a per-axis basis;
- has a straightforward API to control how an array is distributed;
- has basic plotting support for visualization of array distributions;
- separates the array's distribution from the array's data – useful for slicing, reductions, redistribution, broadcasting, and other operations;
- implements distributed random arrays;
- supports `.npy`-like flat-file IO and hdf5 parallel IO (via `h5py`); leverages MPI-based IO parallelism in an easy-to-use and transparent way; and
- supports the distributed array protocol *[protocol]*, which allows independently developed parallel libraries to share distributed arrays without copying, analogous to the PEP-3118 new buffer protocol.

10.3.4 Planned features and roadmap

Near-term features and improvements include:

- MPI-only communication for performance and deployment on clusters and supercomputers;
- array re-distribution capabilities;
- interoperation with Trilinos *[Trilinos]*;
- expanded tutorials, examples, and other introductory material; and
- distributed broadcasting support.

The longer-term roadmap includes:

- Lazy evaluation and deferred computation for latency hiding;
- Integration with other packages *[petsc]* that subscribe to the distributed array protocol *[protocol]*;
- Distributed fancy indexing;
- Out-of-core computations;
- Support for distributed sorting and other non-trivial distributed algorithms; and
- End-user control over communication and temporary array creation, and other performance aspects of distributed computations.

10.3.5 History and funding

Brian Granger started DistArray as a NASA-funded SBIR project in 2008. Enthought picked it up as part of a DOE Phase II SBIR *[SBIR]* to provide a generally useful distributed array package. It builds on NumPy, MPI, MPI4Py, IPython, IPython.parallel, and interfaces with the Trilinos suite of distributed HPC solvers (via PyTrilinos *[Trilinos]*).

This material is based upon work supported by the Department of Energy under Award Number DE-SC0007699.

10.4 DistArray 0.5 release

Mailing List: distarray@googlegroups.com

Documentation: http://distarray.readthedocs.org

License: Three-clause BSD

Python versions: 2.7, 3.3, and 3.4

OS support: *nix and Mac OS X

10.4.1 What is DistArray?

DistArray aims to bring the ease-of-use of NumPy to data-parallel high-performance computing. It provides distributed multi-dimensional NumPy arrays, distributed ufuncs, and distributed IO capabilities. It can efficiently interoperate with external distributed libraries like Trilinos. DistArray works with NumPy and builds on top of it in a flexible and natural way.

10.4.2 0.5 Release

Noteworthy improvements in this release include:

- closer alignment with NumPy's API,
- support for Python 3.4 (existing support for Python 2.7 and 3.3),

- a performance-oriented MPI-only mode for deployment on clusters and supercomputers,
- a way to register user-defined functions to be callable locally on worker processes,
- more consistent naming of sub-packages,
- testing with MPICH2 (already tested against OpenMPI),
- improved and expanded examples,
- installed version testable via `distarray.test()`, and
- performance and scaling improvements.

With this release, DistArray ready for real-world testing and deployment. The project is still evolving rapidly and we appreciate the continued input from the larger scientific-Python community.

10.4.3 Existing features

DistArray:

- supports NumPy-like slicing, reductions, and ufuncs on distributed multidimensional arrays;
- has a client-engine process design – data resides on the worker processes, commands are initiated from master;
- allows full control over what is executed on the worker processes and integrates transparently with the master process;
- allows direct communication between workers, bypassing the master process for scalability;
- integrates with IPython.parallel for interactive creation and exploration of distributed data;
- supports distributed ufuncs (currently without broadcasting);
- builds on and leverages MPI via MPI4Py in a transparent and user-friendly way;
- has basic support for unstructured arrays;
- supports user-controllable array distributions across workers (block, cyclic, block-cyclic, and unstructured) on a per-axis basis;
- has a straightforward API to control how an array is distributed;
- has basic plotting support for visualization of array distributions;
- separates the array's distribution from the array's data – useful for slicing, reductions, redistribution, broadcasting, and other operations;
- implements distributed random arrays;
- supports `.npy`-like flat-file IO and hdf5 parallel IO (via `h5py`); leverages MPI-based IO parallelism in an easy-to-use and transparent way; and
- supports the distributed array protocol *[protocol]*, which allows independently developed parallel libraries to share distributed arrays without copying, analogous to the PEP-3118 new buffer protocol.

10.4.4 Planned features and roadmap

Near-term features and improvements include:

- array re-distribution capabilities;
- lazy evaluation and deferred computation for latency hiding;
- interoperation with Trilinos *[Trilinos]*; and

- distributed broadcasting support.

The longer-term roadmap includes:

- Integration with other packages *[petsc]* that subscribe to the distributed array protocol *[protocol]*;
- Distributed fancy indexing;
- Out-of-core computations;
- Support for distributed sorting and other non-trivial distributed algorithms; and
- End-user control over communication and temporary array creation, and other performance aspects of distributed computations.

10.4.5 History and funding

Brian Granger started DistArray as a NASA-funded SBIR project in 2008. Enthought picked it up as part of a DOE Phase II SBIR *[SBIR]* to provide a generally useful distributed array package. It builds on NumPy, MPI, MPI4Py, IPython, IPython.parallel, and interfaces with the Trilinos suite of distributed HPC solvers (via PyTrilinos *[Trilinos]*).

This material is based upon work supported by the Department of Energy under Award Number DE-SC0007699.

10.5 DistArray 0.6 release

Mailing List: distarray@googlegroups.com

Documentation: http://distarray.readthedocs.org

License: Three-clause BSD

Python versions: 2.7, 3.4, and 3.5

OS support: *nix and Mac OS X

10.5.1 What is DistArray?

DistArray aims to bring the ease-of-use of NumPy to data-parallel high-performance computing. It provides distributed multi-dimensional NumPy arrays, distributed ufuncs, and distributed IO capabilities. It can efficiently interoperate with external distributed libraries like Trilinos. DistArray works with NumPy and builds on top of it in a flexible and natural way.

10.5.2 0.6 Release

Noteworthy improvements in this release include:

- a new website, (http://docs.enthought.com/distarray/), with links to DistArray talks and presentations,

- redistribution for block-distributed (and non-distributed) DistArrays,
- experimental "quickstart" installation scripts,
- an easier API for `Context.apply`
- expanded and improved example notebooks, including a new parallel Gaussian Elimination example by Prashant Mital,
- compatibility with NumPy 1.9,
- expanded TravisCI testing (OS X and Python 3.5),
- logos by Erick Michaud, and
- several bug-fixes and code improvements.

10.5.3 Existing features

DistArray

- supports NumPy-like slicing, reductions, and ufuncs on distributed multidimensional arrays;
- has a client-engine process design – data resides on the worker processes, commands are initiated from master;
- allows full control over what is executed on the worker processes and integrates transparently with the master process;
- allows direct communication between workers, bypassing the master process for scalability;
- integrates with IPython.parallel for interactive creation and exploration of distributed data;
- supports distributed ufuncs (currently without broadcasting);
- builds on and leverages MPI via MPI4Py in a transparent and user-friendly way;
- has basic support for unstructured arrays;
- supports user-controllable array distributions across workers (block, cyclic, block-cyclic, and unstructured) on a per-axis basis;
- has a straightforward API to control how an array is distributed;
- has basic plotting support for visualization of array distributions;
- separates the array's distribution from the array's data – useful for slicing, reductions, redistribution, broadcasting, and other operations;
- implements distributed random arrays;
- supports `.npy`-like flat-file IO and hdf5 parallel IO (via `h5py`); leverages MPI-based IO parallelism in an easy-to-use and transparent way; and
- supports the distributed array protocol *[protocol]*, which allows independently developed parallel libraries to share distributed arrays without copying, analogous to the PEP-3118 new buffer protocol.

10.5.4 Planned features

Planned features include

- array re-distribution capabilities for more distribution types;
- lazy evaluation and deferred computation for latency hiding;
- examples of interoperation with Trilinos *[Trilinos]*;

- distributed broadcasting support.
- integration with other packages *[petsc]* that subscribe to the distributed array protocol *[protocol]*;
- distributed fancy indexing;
- out-of-core computations;
- support for distributed sorting and other non-trivial distributed algorithms; and
- end-user control over communication and temporary array creation, and other performance aspects of distributed computations.

10.5.5 History and funding

Brian Granger started DistArray as a NASA-funded SBIR project in 2008. Enthought picked it up as part of a DOE Phase II SBIR *[SBIR]* to provide a generally useful distributed array package. It builds on NumPy, MPI, MPI4Py, IPython, IPython.parallel, and interfaces with the Trilinos suite of distributed HPC solvers (via PyTrilinos *[Trilinos]*).

This material is based upon work supported by the Department of Energy under Award Number DE-SC0007699.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

[protocol] http://distributed-array-protocol.readthedocs.org/en/rel-0.10.0/

[Trilinos] http://trilinos.org/

[petsc] http://www.mcs.anl.gov/petsc/

[SBIR] http://www.sbir.gov/sbirsearch/detail/410257

[protocol] http://distributed-array-protocol.readthedocs.org/en/rel-0.10.0/

[Trilinos] http://trilinos.org/

[petsc] http://www.mcs.anl.gov/petsc/

[SBIR] http://www.sbir.gov/sbirsearch/detail/410257

[protocol] http://distributed-array-protocol.readthedocs.org/en/rel-0.10.0/

[Trilinos] http://trilinos.org/

[petsc] http://www.mcs.anl.gov/petsc/

[SBIR] http://www.sbir.gov/sbirsearch/detail/410257

[protocol] http://distributed-array-protocol.readthedocs.org/en/rel-0.10.0/

[Trilinos] http://trilinos.org/

[petsc] http://www.mcs.anl.gov/petsc/

[SBIR] http://www.sbir.gov/sbirsearch/detail/410257

d

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z