## Contents

1 Installation .......................................................... 3
2 Getting Started ......................................................... 5
3 History .......................................................................... 7
4 Other Documentation .................................................... 9
   4.1 DistArray API Reference ........................................... 9
   4.2 Building HDF5 and h5py for DistArray ....................... 34
   4.3 Notes on building environment-modules ..................... 35
   4.4 Licence for six.py version 1.5.2 ............................... 37
5 Release Notes ............................................................ 39
   5.1 DistArray 0.2: development release .......................... 39
   5.2 DistArray 0.3: development release .......................... 40
6 Indices and tables ....................................................... 43
Bibliography ............................................................... 45
Python Module Index ..................................................... 47
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.

The project is currently under heavy development and things are changing quickly!

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

Installation

Dependencies for DistArray:

• NumPy
• IPython
• Mpi4Py

Optional dependencies:

• For HDF5 IO: h5py built against a parallel-enabled build of HDF5
• For plotting: matplotlib

Dependencies to build the documentation:

• Sphinx
  • sphinxcontrib.napoleon

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

```
python setup.py install
```

or:

```
python setup.py develop
```

To run the tests, you will need to start an IPython.parallel cluster. You can use `ipcluster`, or you can use the `dacluster` command which comes with DistArray:

```
dacluster start
```

You should then be able to run all the tests with:

```
make test
```

To build this documentation, navigate to the `docs` directory and use the Makefile there. For example, to build the html documentation:

```
make html
```

from the `docs` directory.
Getting Started

To see some initial examples of what distarray can do, check out the examples directory and our tests. More usage examples will be forthcoming as the API stabilizes.
History

DistArray was started by Brian Granger in 2008 and is currently being developed at Enthought by a team led by Kurt Smith, in partnership with Bill Spotz from Sandia’s (Py)Trilinos project and Brian Granger and Min RK from the IPython project.
4.1 DistArray API Reference

4.1.1 distarray Package

4.1.2 error Module

Define error classes.

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

exception distarray.error.DistArrayError
    Bases: exceptions.Exception

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

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

exception distarray.error.MPICommError
    Bases: distarray.error.MPIDistArrayError

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

4.1.3 metadata_utils Module

```python
exception distarray.metadata_utils.GridShapeError
    Bases: exceptions.Exception

exception distarray.metadata_utils.InvalidGridShapeError
    Bases: exceptions.Exception

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.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) –

Returns Contains string distribution type for each dim.

Return type tuple of str

Examples

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

distarray.metadata_utils.normalize_grid_shape(grid_shape, ndims, 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)

4.1.4 testing Module

Functions used for tests.

class distarray.testing.CommNullPasser
Bases: type

Metaclass.

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

class distarray.testing.ContextTestCase(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.

```python
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.

```python
targets = 4
```

```python
class distarray.testing.MpiTestCase(methodName='runTest')
```

```python
Bases: unittest.case.TestCase
Base test class for MPI test cases.
```

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

```python
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.

```python
comm_size = 4
```

```python
class 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.

```python
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.

```python
distarray.testing.check_targets(required, available)
```

If available < required, raise a SkipTest with a nice error message.

```python
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.

```python
distarray.testing.import_or_skip(name)
```

Try importing `name`, raise SkipTest on failure.

```python
Parameters name (str) – Module name to try to import.
```

```python
Returns module – Module object imported by importlib.
```

```python
Return type module object
```

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

### Examples

```python
>>> h5py = import_or_skip('h5py')
>>> h5py.get_config()
<h5py.h5.H5PYConfig at 0x103dd5a78>
```
def raise_typeerror(fn):
    """Decorator for protocol validator functions."
    These functions return (success, err_msg), but sometimes we would rather have an exception.

def 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.

4.1.5 utils Module

Utilities.

def all_equal(iterable):
    """Return True if all elements in iterable are equal."
    Also returns True if iterable is empty.

class count_round_trips:
    """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()

def distarray_random_getstate():

def distarray_random_setstate(state):

def divisors_minmax(n, dmin, dmax):
    """Find the divisors of n in the interval (dmin,dmax]."

def flatten(seq, to_expand=<function list_or_tuple at 0x7faf64acc500>):
    """Flatten a nested sequence."

def get_from_dotted_name(dotted_name):

def hasExactlyOne(iterable):
    """Does iterable have exactly one non-None element?"

def list_or_tuple(seq):

def 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]

def mult_partitions(n, s):
    """Compute the multiplicativ 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_recur(n, s, pd=0)
distarray.utils.multi_for(iterables)
distarray.utils.remove_elements(to_remove, seq)
distarray.utilssanitize_indices(indices)

Check and possibly sanitize indices.

Parameters indices (int, slice, or sequence of ints and slices) – If an int or slice is passed in, it is converted to a 1-tuple.

Returns (‘point’, indices) if all indices are ints, or (‘view’, indices) if some indices are slices.

Return type 2-tuple

Raises TypeError – If indices is not all ints or slices.

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

Currently only implemented for steps of size 1.

Return type slice object

distarray.utils.uid()

4.1.6 Subpackages

apps Package

apps 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()

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.
dist Package

cleanup Module

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

distarray.dist.cleanup.cleanup_all(module_name, prefix)
Connects to all engines and runs cleanup() on them.

distarray.dist.cleanup.clear(view)
Removes all distarray-related modules from engines’ sys.modules.

distarray.dist.cleanup.clear_all()

distarray.dist.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.

context Module

Context objects contain the information required for DistArrays to communicate with LocalArrays.

class distarray.dist.context.Context (client=None, targets=None)
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.

apply (func, args=None, kwargs=None, targets=None)
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.

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

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

close ()

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

distarray.dist.context.empty (distribution, dtype=<type 'float'>)
Create an empty Distarray.

Parameters distribution (Distribution object) –
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.from_shape(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.from_shape(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.local.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.

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

**Return type**  
DistArray

### ones *(distribution, dtype=\(<\text{type} \text{ ‘float’}\>)\)*

Create a Distarray filled with ones.

**Parameters**  
distribution *(Distribution object)* –

**Returns**  
A DistArray distributed as specified, filled with ones.

**Return type**  
DistArray

### 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.local.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 \(<\text{name}>._<\text{rank}>.dnpy\). 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** – The identifier for the group to save the DistArray to (the default is ‘buffer’).

### zeros *(distribution, dtype=\(<\text{type} \text{ ‘float’}\>)\)*

Create a Distarray filled with zeros.

**Parameters**  
distribution *(Distribution object)* –

**Returns**  
A DistArray distributed as specified, filled with zeros.
Return type DistArray

decorators Module

Decorators for defining functions that use DistArrays.

class distarray.dist.decorators.DecoratorBase (fn)
Bases: object

Base class for decorators, handles name wrapping and allows the decorator to take an optional kwarg.

determine_context (args, kwargs)
Determine a context from a functions arguments.

key_and_push_args (args, kwargs, context=None, da_handler=None)
Push a tuple of args and dict of kwargs to the engines. Return a tuple with keys corresponding to args values on the engines. And a dictionary with the same keys and values which are the keys to the input dictionary's values.

This allows us to use the following interface to execute code on the engines:

>>> def foo(*args, **kwargs):
>>>     args, kwargs = _key_and_push_args(args, kwargs)
>>>     exec_str = "remote_foo(*%s, **%s)"
>>>     exec_str %= (args, kwargs)
>>>     context.execute(exec_str)

process_return_value (context, result_key)
Figure out what to return on the Client.

Parameters key (string) – Key corresponding to wrapped function’s return value.

Returns A DistArray (if locally all values are DistArray), a None (if locally all values are None), or else, pull the result back to the client and return it. If all but one of the pulled values is None, return that non-None value only.

Return type Varied

push_fn (context, fn_key, fn)
Push function to the engines.

class distarray.dist.decorators.local (fn)
Bases: distarray.dist.decorators.DecoratorBase

Decorator to run a function locally on the engines.

class distarray.dist.decorators.vectorize (fn)
Bases: distarray.dist.decorators.DecoratorBase

Analogous to numpy.vectorize. Input DistArray’s must all be the same shape, and this will be the shape of the output distarray.

get_ndarray (da, arg_keys)

distarray Module

The Distarray data structure. ‘DistArray’ objects are proxies for collections of LocalArray objects. They are meant to roughly emulate NumPy ndarrays.

class distarray.dist.distarray.DistArray (distribution, dtype=<type 'float'>)
Bases: object
context
dist
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 LocalArrays and a Distribution will be created from it. If distribution is provided, it should accurately reflect the distribution of the existing LocalArrays.

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

global_size
grid_shape
itemsize
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.

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.

**functions Module**

Distributed unfuncs for distributed arrays.

distarray.dist.functions.**absolute** *(a, *args, **kwargs)*
distarray.dist.functions.**arccos** *(a, *args, **kwargs)*
distarray.dist.functions.**arccosh** *(a, *args, **kwargs)*
distarray.dist.functions.**arcsin** *(a, *args, **kwargs)*
distarray.dist.functions.**arcsinh** *(a, *args, **kwargs)*
distarray.dist.functions.**arctan** *(a, *args, **kwargs)*
distarray.dist.functions.**arctanh** *(a, *args, **kwargs)*
distarray.dist.functions.**conjugate** *(a, *args, **kwargs)*
distarray.dist.functions.**cos** *(a, *args, **kwargs)*
distarray.dist.functions.**cosh** *(a, *args, **kwargs)*
distarray.dist.functions.**exp** *(a, *args, **kwargs)*
distarray.dist.functions.**expm1** *(a, *args, **kwargs)*
distarray.dist.functions.**invert** *(a, *args, **kwargs)*
distarray.dist.functions.**log** *(a, *args, **kwargs)*
distarray.dist.functions.**log10** *(a, *args, **kwargs)*
distarray.dist.functions.**log1p** *(a, *args, **kwargs)*
distarray.dist.functions.**negative** *(a, *args, **kwargs)*
distarray.dist.functions.**reciprocal** *(a, *args, **kwargs)*
distarray.dist.functions.**rint** *(a, *args, **kwargs)*
distarray.dist.functions.**sign** *(a, *args, **kwargs)*
distarray.dist.functions.**sin** *(a, *args, **kwargs)*
distarray.dist.functions.**sinh** *(a, *args, **kwargs)*
distarray.dist.functions.**sqrt** *(a, *args, **kwargs)*
distarray.dist.functions.**square** *(a, *args, **kwargs)*
distarray.dist.functions.**tan** *(a, *args, **kwargs)*
distarray.dist.functions.**tanh** *(a, *args, **kwargs)*
distarray.dist.functions.**add** *(a, b, *args, **kwargs)*
distarray.dist.functions.**arctan2** *(a, b, *args, **kwargs)*
distarray.dist.functions.**bitwise_and** *(a, b, *args, **kwargs)*
distarray.dist.functions.bitwise_or(a, b, *args, **kwargs)
distarray.dist.functions.bitwise_xor(a, b, *args, **kwargs)
distarray.dist.functions.divide(a, b, *args, **kwargs)
distarray.dist.functions.floor_divide(a, b, *args, **kwargs)
distarray.dist.functions.fmod(a, b, *args, **kwargs)
distarray.dist.functions.hypot(a, b, *args, **kwargs)
distarray.dist.functions.left_shift(a, b, *args, **kwargs)
distarray.dist.functions.mod(a, b, *args, **kwargs)
distarray.dist.functions.multiply(a, b, *args, **kwargs)
distarray.dist.functions.power(a, b, *args, **kwargs)
distarray.dist.functions.remainder(a, b, *args, **kwargs)
distarray.dist.functions.right_shift(a, b, *args, **kwargs)
distarray.dist.functions.subtract(a, b, *args, **kwargs)
distarray.dist.functions.true_divide(a, b, *args, **kwargs)
distarray.dist.functions.less(a, b, *args, **kwargs)
distarray.dist.functions.less_equal(a, b, *args, **kwargs)
distarray.dist.functions.equal(a, b, *args, **kwargs)
distarray.dist.functions.not_equal(a, b, *args, **kwargs)
distarray.dist.functions.greater(a, b, *args, **kwargs)
distarray.dist.functions.greater_equal(a, b, *args, **kwargs)

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

    dist = 'c'

classmethod from_axis_dim_dicts(axis_dim_dicts)
class \texttt{from\_global\_dim\_dict}(\texttt{glb\_dim\_dict})

\texttt{get\_dimdicts}()

\texttt{owners}(\texttt{idx})

\texttt{class distarray.dist.maps.BlockMap}(\texttt{size}, \texttt{grid\_size})
\texttt{Bases: distarray.dist.maps.MapBase}
\texttt{dist = \textit{b}}

\texttt{class\texttt{method from\_axis\_dim\_dicts}(\texttt{axis\_dim\_dicts})}

\texttt{class\texttt{method from\_global\_dim\_dict}(\texttt{glb\_dim\_dict})}

\texttt{get\_dimdicts}()

\texttt{owners}(\texttt{idx})

\texttt{class distarray.dist.maps.Distribution}(\texttt{context, global\_dim\_data, targets=\textit{None}})
\texttt{Bases: object}

\texttt{Governs the mapping between global indices and process ranks for multi-dimensional objects.}

\texttt{class\texttt{method from\_dim\_data\_per\_rank}(\texttt{context, dim\_data\_per\_rank, targets=\textit{None}})}

Create a Distribution from a sequence of \texttt{dim\_data} tuples.

\texttt{class\texttt{method from\_shape}(\texttt{context, shape, dist=\textit{None}, grid\_shape=\textit{None}, targets=\textit{None}})}

\texttt{get\_dim\_data\_per\_rank}()

\texttt{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 \texttt{checked} version of \texttt{\_getitem\_} or \texttt{\_setitem\_} on LocalArrays.

\texttt{is\_compatible}(\texttt{o})

\texttt{owning\_ranks}(\texttt{idxs})

Returns a list of ranks that may \textit{possibly} own the location in the \texttt{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, \texttt{owning\_ranks()} returns a list of exactly one rank. Otherwise, returns a list of more than one rank.

If the \texttt{idxs} tuple is out of bounds, raises \texttt{IndexError}.

\texttt{owning\_targets}(\texttt{idxs})

Like \texttt{owning\_ranks()} but returns a list of targets rather than ranks.

Convenience method meant for IPython parallel usage.

\texttt{reduce}(\texttt{axes})

Returns a new Distribution reduced along \texttt{axis}, i.e., the new distribution has one fewer dimension than \texttt{self}.

\texttt{class distarray.dist.maps.MapBase}
\texttt{Bases: object}

\texttt{Base class for one-dimensional client-side maps.}

\texttt{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.}

\texttt{Classes that inherit from MapBase must implement the owners() abstractmethod.}

\texttt{is\_compatible}(\texttt{map})
owners(idx)

Returns a list of process IDs in this dimension that might possibly own idx.

 Raises IndexError if idx is out of bounds.

class distarray.dist.maps.NoDistMap(size, grid_size)

Bases: distarray.dist.maps.MapBase

dist = 'n'

classmethod from_axis_dim_dicts(axis_dim_dicts)

classmethod from_global_dim_dict(glb_dim_dict)

glob_dimdicts()

owners(idx)

class distarray.dist.maps.UnstructuredMap(size, grid_size, indices=None)

Bases: distarray.dist.maps.MapBase

dist = 'u'

classmethod from_axis_dim_dicts(axis_dim_dicts)

classmethod from_global_dim_dict(glb_dim_dict)

get_dimdicts()

owners(idx)

distarray.dist.maps.choose_map(dist_type)

Choose a map class given one of the distribution types.

distarray.dist.maps.map_from_global_dim_dict(glob_dim_dict)

Given a global_dim_dict return map.

distarray.dist.maps.map_from_sizes(size, dist_type, grid_size)

Returns an instance of the appropriate subclass of MapBase.

random Module

Emulate numpy.random

class distarray.dist.random.Random(context)

Bases: object

normal(distribution, 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 dis-
tribution of samples influenced by a large number of tiny, random disturbances, each with its own unique
distribution \(^2\).

Parameters

- **loc** (float) – Mean (“centre”) of the distribution.
- **scale** (float) – Standard deviation (spread or “width”) of the distribution.

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 \( \mu \) is the mean and \( \sigma \) the standard deviation. The square of the standard deviation, \( \sigma^2 \), 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 \( x - \sigma \)). This implies that `numpy.random.normal` is more likely to return samples lying close to the mean, rather than those far away.

References

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

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

Returns out – Random values.

Return type DistArray

`randint` *(distribution, 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

- distribution (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` *(distribution)*
Return samples from the “standard normal” distribution.

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.
local Package

constructor Module

distarray.local.construct.init_base_comm(comm)
Sanitize an MPI.comm instance or create one.
distarray.local.construct.init_comm(base_comm, grid_shape)
Create an MPI communicator with a cartesian topology.

error Module

exception distarray.local.error.DistError
Bases: distarray.error.DistArrayError

exception distarray.local.error.DistMatrixError
Bases: distarray.error.DistArrayError

exception distarray.local.error.IncompatibleArrayError
Bases: distarray.error.DistArrayError

exception distarray.local.error.InvalidBaseCommError
Bases: distarray.error.DistArrayError

exception distarray.local.error.InvalidDimensionError
Bases: distarray.error.DistArrayError

exception distarray.local.error.InvalidMapCodeError
Bases: distarray.error.DistArrayError

exception distarray.local.error.NullCommError
Bases: distarray.error.DistArrayError

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:

```
“dim_data” [tuple of dict] One dictionary per array dimension; see the Distributed Array Protocol for

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.

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.local.format.magic(major, minor, prefix=<MagicMock name='mock.asbytes()' id='140391284290896'>)

Return the magic string for the given file format version.

Parameters major (int in [0, 255]) –
Returns magic
Return type str
Raises ValueError – if the version cannot be formatted.

distarray.local.format.read_array_header_1_0(fp)

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.

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.local.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.local.format.read_magic(fp)

Read the magic string to get the version of the file format.

Returns

• major (int)

• minor (int)
```
distarray.local.format.write_localarray (fp, arr, 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.
• arr (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.

localarray Module

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

Object which provides access to global indexing on LocalArrays.

checked_getitem (global_inds)
checked_setitem (global_inds, value)
global_to_local (*global_ind)
local_to_global (*local_ind)

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

class distarray.local.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 \((\text{other})\)  
Return a version of self that has shape, dist and grid_shape like \text{other}.

astype \((\text{newdtype})\)  
Return a copy of this LocalArray with a new underlying dtype.

cart_coords  
comm  
comm_rank  
comm_size  
compatibility_hash()  
coords_from_rank \((\text{rank})\)  
copy()  
Return a copy of this LocalArray.

dim_data  
dist  
dtype  
fill \((\text{scalar})\)  
classmethod from_distarray \((\text{comm}, \text{obj})\)  
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  
get_localarray()  
global_from_local \((*\text{local\_ind})\)  
global_limits \((\text{dim})\)  
global_shape  
global_size  
grids_shape  
itemsize  
local_data  
local_from_global \((*\text{global\_ind})\)  
local_shape  
local_size  
local_view \((\text{dtype}=\text{None})\)  
nbytes
class distarray.local.localarray.LocalArrayBinaryOperation(numpy_ufunc)
Bases: object
class distarray.local.localarray.LocalArrayUnaryOperation(numpy_ufunc)
Bases: object
distarray.local.localarray.arecompatible(a, b)
Do these arrays have the same compatibility hash?
distarray.local.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.local.localarray.empty(distribution, dtype=<type 'float'>)
Create an empty LocalArray.
distarray.local.localarray.empty_like(arr, dtype=None)
Create an empty LocalArray with a distribution like arr.
distarray.local.localarray.fromfunction(function, distribution, **kwargs)
distarray.local.localarray.fromndarray_like(ndarray, like_arr)
Create a new LocalArray like like_arr with buffer set to ndarray.
distarray.local.localarray.get_printoptions()
distarray.local.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.local.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 hSpy / hdf5.

distarray.local.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.

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

Return type `LocalArray`

distarray.local.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.

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

Return type `LocalArray` or `None`

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

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

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

distarray.local.localarray.ndenumerate (arr)
Create a LocalArray filled with ones.
distarray.local.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.local.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** – The identifier for the group to save the LocalArray to (the default is ‘buffer’).

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

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

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

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

distarray.local.localarray.zeros (distribution, dtype=<type ‘float’>)
Create a LocalArray filled with zeros.

distarray.local.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.local.maps.BlockCyclicMap (global_size, grid_size, grid_rank, start, block_size)
Bases: distarray.local.maps.MapBase
One-dimensional block cyclic map class.

dim_dict
dist = ‘c’
global_from_local (lidx)
global_iter
global_from_global (gidx)
size
class distarray.local.maps.BlockMap (global_size, grid_size, grid_rank, start, stop)
    Bases: distarray.local.maps.MapBase
    One-dimensional block map class.
    dim_dict
dist = 'b'
global_from_local(lidx)
global_iter
local_from_global(gidx)

size
class distarray.local.maps.CyclicMap (global_size, grid_size, grid_rank, start)
    Bases: distarray.local.maps.MapBase
    One-dimensional cyclic map class.
    dim_dict
dist = 'c'
global_from_local(lidx)
global_iter
local_from_global(gidx)

size
class distarray.local.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
grid_shape
local_from_global(*global_ind)
    Given global_ind indices, translate into local indices.
class distarray.local.maps.MapBase

    Bases: object

    Base class for all one dimensional Map classes.

class distarray.local.maps.UnstructuredMap

    Bases: distarray.local.maps.MapBase

    One-dimensional unstructured map class.

    dim_dict
dist = 'u'
global_from_local(lidx)
global_iter
local_from_global(gidx)

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

distarray.local.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.local.mpiutils.mpi_type_for_ndarray(a)

proxyize Module

class distarray.local.proxyize.Proxyize(context_key)

    Bases: object

    next_name()

    set_state(state)

    str_counter()
**random Module**

distarray.local.random.beta \((a, b, \text{distribution=None})\)
distarray.local.random.label_state \((\text{comm})\)
  
  Label/personalize the random generator state for the local rank.

distarray.local.random.normal \((\text{loc=0.0, scale=1.0, distribution=None})\)
distarray.local.random.rand \((\text{distribution=None})\)
distarray.local.random.randint \((\text{low, high=None, distribution=None})\)
distarray.local.random.randn \((\text{distribution=None})\)

**plotting Package**

**plotting Package**

Plotting functions for distarrays.

**plotting Module**

Plotting functions for distarrays.

distarray.plotting.plotting.cmap_discretize \((\text{cmap, N})\)
  
  Create a discrete colormap from the continuous colormap cmap.

  **Parameters**
  
  • cmap \((\text{colormap instance, or string})\) – The continuous colormap, as object or name, to make discrete. For example, matplotlib.cm.jet, or ‘jet’.
  
  • N \((\text{int})\) – The number of discrete colors desired.

  **Returns**
  
  The desired discrete colormap.

  **Return type** colormap

**Example**

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

distarray.plotting.plotting.create_discrete_colormaps \((\text{num_values})\)
  
  Create colormap objects for a discrete colormap.

  **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
The DistArray documentation, Release 0.3.0

```
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.

## 4.2 Building HDF5 and h5py for DistArray

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.

### 4.2.1 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

### 4.2.2 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

### 4.3 Notes on building 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.
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-module-path to override default
    configure: WARNING: will use VERSIONPATH=/usr/local/Modules/versions : rerun configure using --with-version-path to override default

    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!

### 4.3.1 References

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


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


### 4.4 Licence for six.py version 1.5.2

Copyright (c) 2010-2014 Benjamin Peterson

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
Release Notes

5.1 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


5.2 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

5.2.1 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.
5.2.2 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.

5.2.3 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.
5.2.4 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

5.2.5 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]).
Indices and tables

- genindex
- modindex
- search
[protocol] http://distributed-array-protocol.readthedocs.org/en/rel-0.10.0/
Python Module Index

```
d
distarray.__init__, 9
distarray.apps.__init__, 13
distarray.apps.dacluster, 13
distarray.dist.__init__, 14
distarray.dist.cleanup, 14
distarray.dist.context, 14
distarray.dist.decorators, 17
distarray.dist.distarray, 17
distarray.dist.functions, 19
distarray.dist.ipython_utils, 20
distarray.dist.maps, 20
distarray.dist.random, 22
distarray.error, 9
distarray.local.__init__, 24
distarray.local.construct, 24
distarray.local.error, 24
distarray.local.format, 24
distarray.local.localarray, 26
distarray.local.maps, 30
distarray.local.mpiutils, 32
distarray.local.proxyize, 32
distarray.local.random, 33
distarray.metadata_utils, 9
distarray.plotting.__init__, 33
distarray.plotting.plotting, 33
distarray.testing, 10
distarray.utils, 12
```