Distributed Array Protocol Documentation

Release 0.10.0

Jeff Daily **Brian Granger** **Robert Grant**
Min Ragan-Kelley **Mark Kness** **Kurt Smith**
Bill Spotz

August 13, 2015

Contents

CHAPTER 1

Distributed Array Protocol

1.1 Overview

The Distributed Array Protocol (DAP) is a process-local protocol that allows two subscribers, called the "producer" and the "consumer" or the "exporter" and the "importer", to communicate the essential data and metadata necessary to share a distributed-memory array between them. This allows two independently developed components to access, modify, and update a distributed array without copying. The protocol formalizes the metadata and buffers involved in the transfer, allowing several distributed array projects to collaborate, facilitating interoperability. By not copying the underlying array data, the protocol allows for efficient sharing of array data.

The DAP is intended to build on the concepts and implementation of the existing PEP-3118 buffer protocol [1], and uses PEP-3118 buffers (and subscribing Python objects such as memoryviews and NumPy arrays) as components.

This version of the DAP is defined for the Python language. Future versions of this protocol may provide definitions in other languages.

1.2 Usecases

Major usecases supported by the protocol include:

- Sharing large amounts of array data without copying.
- Block, cyclic, and block-cyclic distributions for structured arrays.
- Padded block-distributed arrays for finite differencing applications.
- Unstructured distributions for arbitrary mappings between global indices and local data.
- Multi-dimensional arrays.
- Different distributions for each array dimension.
- Dense (structured) and sparse (unstructured) arrays.
- Compatibility with array views and slices.

1.3 Sources

The primary sources and inspiration for the DAP are:

[1] Revising the Buffer Protocol. http://www.python.org/dev/peps/pep-3118/

- NumPy [2] and the Revised Buffer Protocol [4]
- Trilinos [3] and PyTrilinos [4]
- Global Arrays [5] and Global Arrays in NumPy (GAiN) [6]
- The Chapel [7], X10 [8], and HP-Fortran [9] languages
- Distributed array implementation in the Julia [10] language

1.4 Definitions

process A **process** is the basic unit of execution and is equivalent to a conventional OS process. Each process has an address space, has one or more namespaces that contain objects, and is able to communicate with other processes to send and receive data. Note that the protocol does not require any inter-process communication and makes no assumptions regarding communication libraries.

process rank An integer label that uniquely identifies a process. Ranks are assigned contiguously from the range `0 ... N-1` for `N` processes.

process grid The **process grid** is an N-dimensional Cartesian grid. Each coordinate uniquely identifies a process, and the process grid maps process ranks to grid coordinates. Process ranks are assigned to their corresponding grid coordinate in "C-order", i.e., the last index varies fastest when iterating through coordinates in rank order. The product of the number of processes in each dimension in the process grid shall be equal to the total number of processes.

For example, for an `N` by `M` process grid over `N * M` processes with ranks `0, 1, ..., (N*M)-1`, process grid coordinate `(i,j)` corresponds to the process with rank `i*M + j`.

(Note that the protocol's process grid is compatible with MPI's `MPI_Cart_create()` command, and the MPI standard guarantees that Cartesian process coordinates are always assigned to ranks in the same way and are "C-order" by default [11]. The protocol makes no assumption about which underlying communication library is being used, nor does it require subscribing packages to implement a communication layer.)

distributed array A single logical array of arbitrary dimensionality that is divided among multiple processes.

A distributed array has both a global and a local index space for each dimension, and a mapping between the two index spaces.

local index A local index specifies a location in an array's data at the process level.

global index A global index specifies a location in the array's data as if the array were not distributed.

map A map object provides two functionalities: the first is the ability to translate a global index into a process identifier and a local index on that process; the second is the ability to provide the global index that corresponds to a given local index.

boundary padding Padding indices in a local array that indicate which indices are part of the logical *boundary* of the entire domain. These are physical or real boundaries and correspond to the elements or indices that are involved with the physical system's boundary conditions in a PDE application, for example. These elements are included in a distributed dimension's `'size'`.

[2] NumPy. http://www.numpy.org/
[3] Trilinos. http://trilinos.sandia.gov/
[4] PyTrilinos. http://trilinos.sandia.gov/packages/pytrilinos/
[5] Global Arrays. http://hpc.pnl.gov/globalarrays/
[6] Global Arrays in NumPy. http://www.pnnl.gov/science/highlights/highlight.asp?id=1043
[7] Chapel. http://chapel.cray.com/
[8] X10. http://x10-lang.org/
[9] High Perfomance Fortran. http://dacnet.rice.edu/
[10] Julia. http://docs.julialang.org
[11] MPI-2.2 Standard: Virtual Topologies. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node165.htm#Node165

communication padding Padding indices that are shared logically with a neighboring local array. These padding regions are used often in finite differencing applications and reserve room for communication with neighboring arrays when data updates are required. Each of these shared elements are only counted once toward the `'size'` of each distributed dimension, so the total `'size'` of a dimension will less than or equal to the sum of the sizes all local buffers.

1.5 Exporting a Distributed Array

A "producer" object that subscribes to the DAP shall provide a method named `__distarray__` that, when called by a consumer, returns a dictionary with three keys: `'__version__'`, `'buffer'`, and `'dim_data'`.

The value associated with the `'__version__'` key shall be a string of the form `'major.minor.patch'`, as described in the Semantic Versioning specification [12] and PEP-440 [13]. As specified in Semantic Versioning, versions of the protocol that differ in the minor version number shall be backwards compatible; versions that differ in the major version number may break backwards compatibility.

The value associated with the `'buffer'` key shall be a Python object that is compatible with the PEP-3118 buffer protocol and contains the data for a local section of a distributed array.

The value for the `'dim_data'` key shall be a tuple of dictionaries, called "dimension dictionaries", containing one dictionary for each dimension of the distributed array. The zeroth dictionary in `'dim_data'` shall describe the zeroth dimension of the array, the first dictionary shall describe the first dimension, and so on for each dimension in succession. These dictionaries include all metadata required to specify a distributed array's distribution. The `'dim_data'` tuple may be empty, indicating a zero-dimensional array. The number of elements in the `'dim_data'` tuple must match the number of dimensions of the associated buffer object.

1.6 Dimension Dictionaries

All dimension dictionaries shall have a `'dist_type'` key with a value of type string. The `dist_type` of a dimension specifies the kind of distribution for that dimension.

The following dist_types are currently supported:

name	dist_type	required keys	optional keys
block	'b'	common, 'start', 'stop'	'padding', 'periodic'
cyclic	'c'	common, 'start'	'block_size'
unstructured	'u'	common, 'indices'	'one_to_one'

where "common" represents the keys common to all dist_types: `'dist_type'`, `'size'`, `'proc_grid_size'`, and `'proc_grid_rank'`.

Other `dist_types` may be added in future versions of the protocol.

1.6.1 Required key-value pairs

All dimension dictionaries (regardless of distribution type) must define the following key-value pairs:

- `'dist_type'`: `{'b', 'c', 'u'}`.

  The distribution type; the primary way to determine the kind of distribution for this dimension.

[12] Semantic Versioning 2.0.0. http://semver.org/
[13] PEP 440: Version Identification and Dependency Specification. http://www.python.org/dev/peps/pep-0440/

- `'size'` : `int`, greater than or equal to 0.

  Total number of global array elements along this dimension.

  Indices considered "communication padding" *are not* counted towards this value; indices considered "boundary padding" *are* counted towards this value. More explicitly, to calculate the `size` along a particular dimension, one can sum the result of the function `num_owned_indices` (in the provided `utils.py` or in this document's appendix) run on the appropriate dimension dictionary on every process.

- `'proc_grid_size'` : `int`, greater than or equal to 1.

  The total number of processes in the process grid in this dimension. Necessary for computing the global / local index mapping, etc.

  Constraint: the product of all `'proc_grid_size'`s for all dimensions shall equal the total number of processes.

- `'proc_grid_rank'` : `int`, greater than or equal to 0, less than `'proc_grid_size'`.

  The rank of the process for this dimension in the process grid. This information allows the consumer to determine where the neighbor sections of an array are located.

  The mapping of process rank to process grid coordinates is assumed to be row major. For an `N` by `M` process grid over `N * M` processes with ranks `0, 1, ..., (N*M)-1`, process grid coordinate `(i,j)` corresponds to the process with rank `i*M + j`. This generalizes in the conventional row-major way.

1.6.2 Distribution-type specific key-value pairs

The remaining key-value pairs in each dimension dictionary depend on the `dist_type` and are described below.

block (`dist_type` is `'b'`)

- `start` : `int`, greater than or equal to zero.

  The start index (inclusive and 0-based) of the global index space available on this process.

- `stop` : `int`, greater than the `start` value, less than or equal to the `size` value.

  The stop index (exclusive, as in standard Python indexing) of the global index space available on this process.

  For a block-distributed dimension without communication padding, adjacent processes as determined by the dimension dictionary's `proc_grid_rank` field shall have adjacent global index ranges. More explicitly, for two processes `a` and `b` with grid ranks `i` and `i+1` respectively, the `stop` of `a` shall be equal the `start` of `b`. With communication padding present, the stop of `a` may be greater than the `start` of `b`.

  Processes may contain differently-sized global index ranges; this is sometimes called an "irregular block distribution".

  For every block-distributed dimension `i`, `stop - start` must be equal to `buffer.shape[i]`.

- `padding` : 2-tuple of `int`, each greater than or equal to zero. Optional.

  If a value represents communication padding width, it must be less than or equal to the number of indices owned by the neighboring process.

  The padding tuple describes the width of the padding region at the beginning and end of a buffer in a particular dimension. Padding represents extra allocation for an array, but padding values are in some sense not "owned" by the local array and are reserved for other purposes.

  For the dimension dictionary with `proc_grid_rank == 0`, the first element in `padding` is the width of the boundary padding; this is extra allocation reserved for boundary logic in applications that need it. For the

dimension dictionary with `proc_grid_rank == proc_grid_size-1`, the second element in `padding` is the width of the boundary padding. All other `padding` tuple values are for communication padding and represent extra allocation reserved for communication between processes. Every communication padding width must equal its counterpart on its neighboring process; more specifically, the "right" communication padding on rank `i` in a 1D grid must equal the "left" communication padding on rank `i+1`.

For example, consider a one-dimensional block-distributed array distributed over four processes. Let its left boundary padding width be 4, its right boundary padding width be 0 and its communication padding widths be (1,) (1, 2), (2, 3), and (3,). The padding tuple for the local array on each rank would be:

proc_grid_rank	0	1	2	3
padding	(4, 1)	(1, 2)	(2, 3)	(3, 0)

If the value associated with `padding` is the tuple `(0,0)` (the default), this indicates the local array is not padded in this dimension.

- `periodic` : `bool`, optional.

  Indicates whether this dimension is periodic. When not present, indicates this dimension is not periodic, equivalent to a value of `False`.

cyclic (`dist_type` is `'c'`)

- `start` : `int`, greater than or equal to zero.

  The start index (inclusive, 0-based) of the global index space available on this process.

  The cyclic distribution is what results from assigning global indices (or contiguous blocks of indices when `block_size` is greater than one) to processes in round-robin fashion. When `block_size` equals one, a Python slice formed from the `start`, `size`, and `proc_grid_size` values selects the global indices that are owned by this local array.

- `block_size` : `int`, greater than or equal to one. Optional.

  Indicates the size of contiguous blocks of indices for this dimension. If absent, equivalent to the case when `block_size` is present and equal to one.

  If `block_size == 1` (the default), this specifies the "true" cyclic distribution as described in the ScaLAPACK documentation [14]. If `block_size == ceil(size / proc_grid_size)`, this distribution is equivalent to an evenly-distributed block distribution. If `1 < block_size < size // proc_grid_size`, then this specifies a distribution sometimes called "block-cyclic" [16] [15].

  Block-cyclic is a generalization of (evenly-distributed) block and cyclic distribution types. It can be thought of as as a cyclic distribution with contiguous blocks of global indices (rather than single indices) distributed in a round robin fashion.

  Note that since this protocol allows for block-distributed dimensions with irregular numbers of indices on each process, not all 'block'-distributed dimensions describable by this protocol can be represented as 'cyclic' with the `block_size` key.

unstructured (`dist_type` is `'u'`)

- `indices` : buffer (or buffer-compatible) of `int`.

  Global indices available on this process.

[14] ScaLAPACK Users' Guide: The Two-dimensional Block-Cyclic Distribution. http://netlib.org/scalapack/slug/node75.html

[15] Parallel ESSL Guide and Reference: Block-Cyclic Distribution over Two-Dimensional Process Grids. http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=%2Fcom.ibm.cluster.pessl.v4r2.pssl100.doc%2Fam6gr_dvtdpg.htm

The only constraint that applies to the `indices` buffer is that the values are locally unique. The indices values are otherwise unconstrained: they can be negative, unordered, and non-contiguous.

- `one_to_one` : `bool`, optional.

  If not present, shall be equivalent to being present with a `False` value.

  If `False`, indicates that some global indices may be duplicated in two or more local `indices` buffers.

  If `True`, a global index shall be located in exactly one local `indices` buffer.

1.6.3 Dimension dictionary aliases

The following aliases are provided for convenience. Only one is provided in the current version of this protocol, but more may be added in future versions.

Empty dimension dictionary

An empty dimension dictionary in dimension `i` of `'dim_data'`, will be interpreted as the following:

```
{'dist_type': 'b',
 'proc_grid_rank': 0,
 'proc_grid_size': 1,
 'start': 0,
 'stop': buf.shape[i],
 'size': buf.shape[i]}
```

where `buf` is the associated buffer object.

This is intended to be a shortcut for defining undistributed dimensions.

1.6.4 General comments

Empty local buffers

It shall be possible for one or more local array sections to contain no data. This is supported by the protocol and is not an invalid state. These situations may arise explicitly or when downsampling or slicing a distributed array.

The following properties of a dimension dictionary imply an empty local buffer:

- With any `dist_type`: `size == 0`
- With the `'b'` or `'c'` `dist_type`: `start == size`
- With the `'b'` `dist_type`: `start == stop`
- With the `'u'` `dist_type`: `len(indices) == 0`

Undistributed dimensions

A dimension with `proc_grid_size == 1` is essentially undistributed; it is "distributed" over a single process. Block-distributed dimensions with `proc_grid_size == 1` and with the `periodic` and `padding` keys present are valid. `periodic == True` and nonzero `padding` values indicate this array is periodic on one processor and has associated padding regions.

Global array size

The global number of elements in an array is the product of the values of `'size'` in the dimension dictionaries, or `1` if the `'dim_data'` sequence is empty. In Python syntax, this would be `reduce(operator.mul, global_shape, 1)` where `global_shape` is a Python sequence of integers such that `global_shape[i]` is the value of `'size'` in the dimension dictionary for dimension `i`. If `global_shape` is an empty sequence, the result of the reduction above is `1`, indicating the distributed array is a zero-dimensional scalar.

Identical `'dim_data'` along an axis

If `dim_data` is the tuple of dimension dictionaries for a process and `rank = dim_data[i]['proc_grid_rank']` for some dimension `i`, then all processes with the same `rank` for dimension `i` must have the same values for other keys in their respective dimension dictionaries. Essentially, this says that dimension dictionary `dim_data[i]` is identical for all processes that have the same value for `dim_data[i]['proc_grid_rank']`. The only possible exception to this is the `padding` tuple, which may have different values on edge processes due to boundary padding.

1.7 References

CHAPTER 2

Examples

2.1 Block, Block

Assume we have a process grid with 2 rows and 1 column, and we have a 2x10 array `a` distributed over it. Let `a` be a two-dimensional array with a block distribution in both dimensions. Note that since the `proc_grid_size` of the first dimension is `1`, it is essentially undistributed. Because of this, having a cyclic `dist_type` for this dimension would be equivalent.

In process 0:

```
>>> distbuffer = a0.__distarray__()
>>> distbuffer.keys()
['__version__', 'buffer', 'dim_data']
>>> distbuffer['__version__']
'0.10.0'
>>> distbuffer['buffer']
array([ 0.2,  0.6,  0.9,  0.6,  0.8,  0.4,  0.2,  0.2,  0.3,  0.5])
>>> distbuffer['dim_data']
({'size': 2,
  'dist_type': 'b',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'start': 0,
  'stop': 1},
 {'size': 10,
  'dist_type': 'b',
  'proc_grid_rank': 0,
  'proc_grid_size': 1,
  'start': 0,
  'stop': 10})
```

In process 1:

```
>>> distbuffer = a1.__distarray__()
>>> distbuffer.keys()
['__version__', 'buffer', 'dim_data']
>>> distbuffer['__version__']
'0.10.0'
>>> distbuffer['buffer']
array([ 0.9,  0.2,  1. ,  0.4,  0.5,  0. ,  0.6,  0.8,  0.6,  1. ])
>>> distbuffer['dim_data']
({'size': 2,
  'dist_type': 'b',
```

```
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'start': 1,
  'stop': 2},
 {'size': 10,
  'dist_type': 'b',
  'proc_grid_rank': 0,
  'proc_grid_size': 1,
  'start': 0,
  'stop': 10})
```

2.2 Block with padding

Assume we have a process grid with 2 processes, and we have an 18-element array `a` distributed over it. Let `a` be a one-dimensional array with a block-padded distribution for its 0th (and only) dimension.

Since the `'padding'` for each process is `(1, 1)`, the local array on each process has one element of padding on the left and one element of padding on the right. Since each of these processes is at one edge of the process grid (and the array has no `'periodic'` dimensions), the "outside" element on each local array is an example of "boundary padding", and the "inside" element on each local array is an example of "communication padding". Note that the `'size'` of the distributed array is not equal to the combined buffer sizes of `a0` and `a1` , since communication padding is not counted toward `'size'` (though the boundary padding is).

For this example, the distribution of global indices (with 'B' representing boundary padding and 'C' representing communication padding) is as follows:

```
Process 0: B 1 2 3 4 5 6 7 8 C
Process 1:                 C 9 10 11 12 13 14 15 16 B
```

The 'B' element on process 0 occupies global index 0, and the 'B' element on process 1 occupies global index 17. Each 'B' element counts towards the array's `'size'`. The communication elements on each process overlap with a data element on the other process to indicate which data elements these communication elements are meant to communicate with.

The protocol data structure on each process is as follows.

In process 0:

```
>>> distbuffer = a0.__distarray__()
>>> distbuffer.keys()
['__version__', 'buffer', 'dim_data']
>>> distbuffer['__version__']
'0.10.0'
>>> distbuffer['buffer']
array([ 0.2,  0.6,  0.9,  0.6,  0.8,  0.4,  0.2,  0.2,  0.3,  0.9])
>>> distbuffer['dim_data']
({'size': 18,
  'dist_type': 'b',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'start': 0,
  'stop': 10,
  'padding': (1, 1)})
```

In process 1:

```
>>> distbuffer = a1.__distarray__()
>>> distbuffer.keys()
['__version__', 'buffer', 'dim_data']
>>> distbuffer['__version__']
'0.10.0'
>>> distbuffer['buffer']
array([ 0.3,  0.9,  0.2,  1. ,  0.4,  0.5,  0. ,  0.6,  0.8,  0.6])
>>> distbuffer['dim_data']
({'size': 18,
  'dist_type': 'b',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'start': 8,
  'stop': 18,
  'padding': (1, 1)})
```

2.3 Unstructured

Assume we have a process grid with 3 rows, and we have a size 30 array `a` distributed over it. Let `a` be a one-dimensional unstructured array with 7 elements on process 0, 3 elements on process 1, and 20 elements on process 2.

On all processes:

```
>>> distbuffer = local_array.__distarray__()
>>> distbuffer.keys()
['__version__', 'buffer', 'dim_data']
>>> distbuffer['__version__']
'0.10.0'
>>> len(distbuffer['dim_data']) == 1  # one dimension only
True
```

In process 0:

```
>>> distbuffer['buffer']
array([0.7,  0.5,  0.9,  0.2,  0.7,  0.0,  0.5])
>>> distbuffer['dim_data']
({'size': 30,
  'dist_type': 'u',
  'proc_grid_rank': 0,
  'proc_grid_size': 3,
  'indices': array([19, 1, 0, 12, 2, 15, 4])},)
```

In process 1:

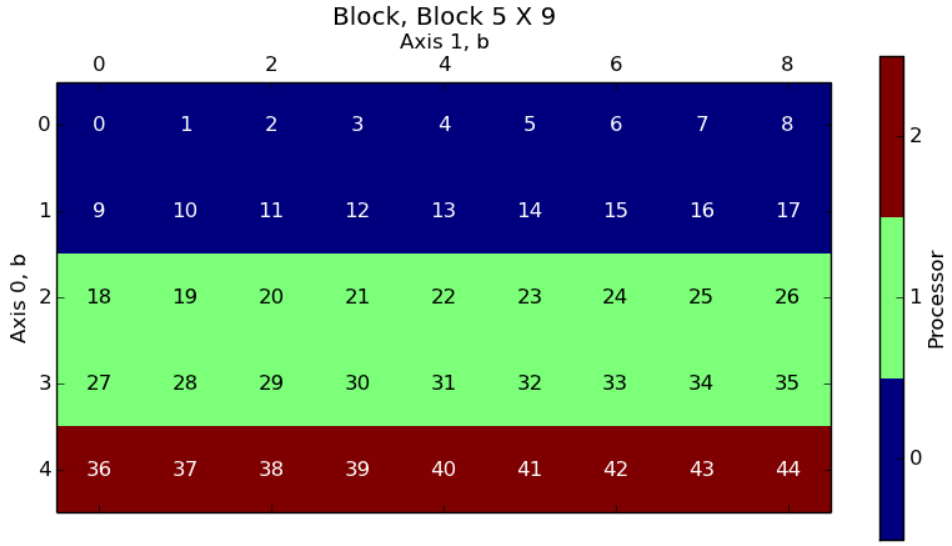
```
>>> distbuffer['buffer']
array([0.1,  0.5,  0.9])
>>> distbuffer['dim_data']
({'size': 30,
  'dist_type': 'u',
  'proc_grid_rank': 1,
  'proc_grid_size': 3,
  'indices': array([6, 13, 3])},)
```

In process 2:

```
>>> distbuffer['buffer']
array([ 0.1,  0.8,  0.4,  0.8,  0.2,  0.4,  0.4,  0.3,  0.5,  0.7,
        0.4,  0.7,  0.6,  0.2,  0.8,  0.5,  0.3,  0.8,  0.4,  0.2])
>>> distbuffer['dim_data']
({'size': 30,
  'dist_type': 'u',
  'proc_grid_rank': 2,
  'proc_grid_size': 3,
  'indices': array([10, 25,  5, 21,  7, 18, 11, 26, 29, 24, 23, 28, 14,
             20,  9, 16, 27,  8, 17, 22])},)
```

2.4 Block, Block

A (5 X 9) array, with a Block, Block ('b' X 'b') distribution over a (3 X 1) process grid.



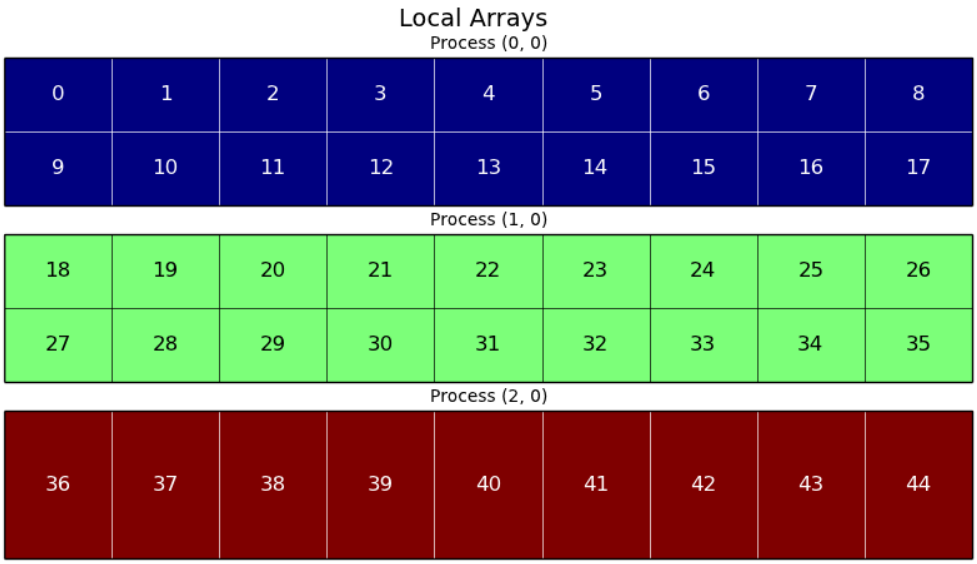
The full (undistributed) array:

```
>>> full_array
array([[  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.],
       [ 18.,  19.,  20.,  21.,  22.,  23.,  24.,  25.,  26.],
       [ 27.,  28.,  29.,  30.,  31.,  32.,  33.,  34.,  35.],
       [ 36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,  44.]])
```

In all processes, we have:

```
>>> distbuffer = local_array.__distarray__()
>>> distbuffer.keys()
['buffer', 'dim_data', '__version__']
>>> distbuffer['__version__']
'0.10.0'
```

The local arrays, on each separate engine:


Local Arrays
Process (0, 0)
0 1 2 3 4 5 6 7 8
9 10 11 12 13 14 15 16 17
Process (1, 0)
18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35
Process (2, 0)
36 37 38 39 40 41 42 43 44

In process (0, 0):

```
>>> distbuffer['buffer']
array([[  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 0,
  'proc_grid_size': 3,
  'size': 5,
  'start': 0,
  'stop': 2},
 {'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 0,
  'proc_grid_size': 1,
  'size': 9,
  'start': 0,
  'stop': 9})
```

In process (1, 0):

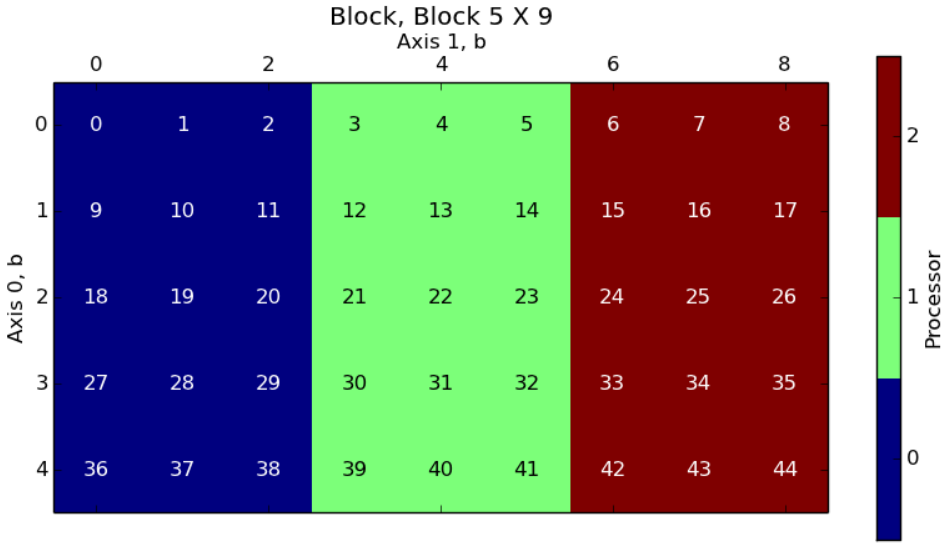
```
>>> distbuffer['buffer']
array([[ 18.,  19.,  20.,  21.,  22.,  23.,  24.,  25.,  26.],
       [ 27.,  28.,  29.,  30.,  31.,  32.,  33.,  34.,  35.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 1,
  'proc_grid_size': 3,
  'size': 5,
  'start': 2,
  'stop': 4},
 {'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 0,
  'proc_grid_size': 1,
  'size': 9,
  'start': 0,
  'stop': 9})
```

In process (2, 0):

```
>>> distbuffer['buffer']
array([[ 36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,  44.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 2,
  'proc_grid_size': 3,
  'size': 5,
  'start': 4,
  'stop': 5},
 {'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 0,
  'proc_grid_size': 1,
  'size': 9,
  'start': 0,
  'stop': 9})
```

2.5 Block, Block

A (5 X 9) array, with a Block, Block ('b' X 'b') distribution over a (1 X 3) process grid.



The full (undistributed) array:

```
>>> full_array
array([[  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.],
       [ 18.,  19.,  20.,  21.,  22.,  23.,  24.,  25.,  26.],
       [ 27.,  28.,  29.,  30.,  31.,  32.,  33.,  34.,  35.],
       [ 36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,  44.]])
```

In all processes, we have:

```
>>> distbuffer = local_array.__distarray__()
>>> distbuffer.keys()
['buffer', 'dim_data', '__version__']
>>> distbuffer['__version__']
'0.10.0'
```

The local arrays, on each separate engine:

Local Arrays

Process (0, 0)

0	1	2
9	10	11
18	19	20
27	28	29
36	37	38

Process (0, 1)

3	4	5
12	13	14
21	22	23
30	31	32
39	40	41

Process (0, 2)

6	7	8
15	16	17
24	25	26
33	34	35
42	43	44

In process (0, 0):

```
>>> distbuffer['buffer']
array([[  0.,   1.,   2.],
       [  9.,  10.,  11.],
       [ 18.,  19.,  20.],
       [ 27.,  28.,  29.],
       [ 36.,  37.,  38.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 0,
  'proc_grid_size': 1,
  'size': 5,
  'start': 0,
  'stop': 5},
 {'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 0,
  'proc_grid_size': 3,
  'size': 9,
  'start': 0,
  'stop': 3})
```

In process (0, 1):

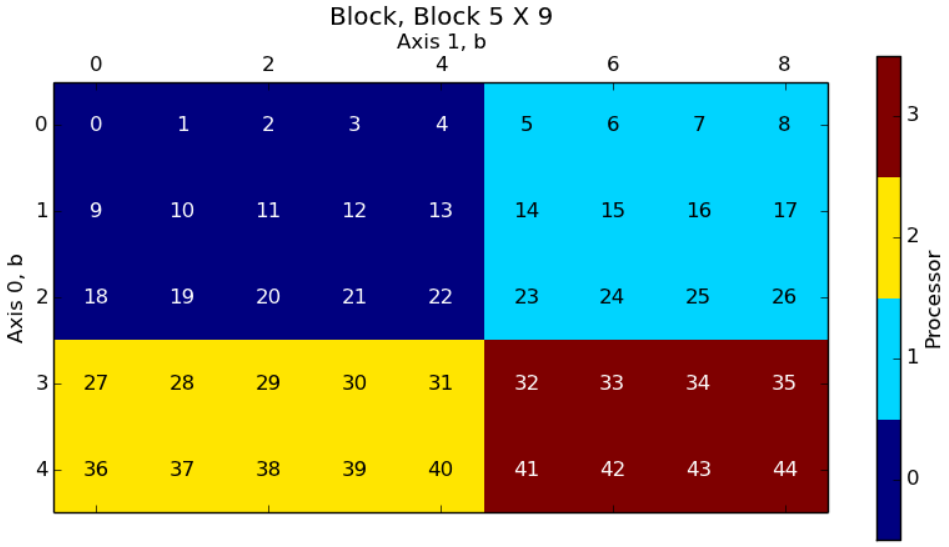
```
>>> distbuffer['buffer']
array([[  3.,   4.,   5.],
       [ 12.,  13.,  14.],
       [ 21.,  22.,  23.],
       [ 30.,  31.,  32.],
       [ 39.,  40.,  41.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 0,
  'proc_grid_size': 1,
  'size': 5,
  'start': 0,
  'stop': 5},
 {'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 1,
  'proc_grid_size': 3,
  'size': 9,
  'start': 3,
  'stop': 6})
```

In process (0, 2):

```
>>> distbuffer['buffer']
array([[  6.,   7.,   8.],
       [ 15.,  16.,  17.],
       [ 24.,  25.,  26.],
       [ 33.,  34.,  35.],
       [ 42.,  43.,  44.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 0,
  'proc_grid_size': 1,
  'size': 5,
  'start': 0,
```

2.6 Block, Block

A (5 X 9) array, with a Block, Block ('b' X 'b') distribution over a (2 X 2) process grid.



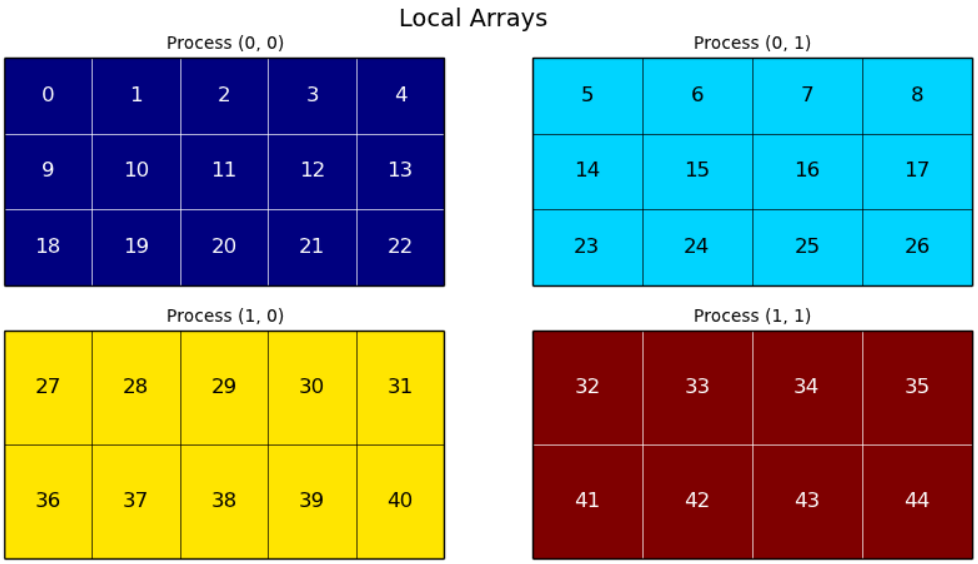
The full (undistributed) array:

```
>>> full_array
array([[  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.],
       [ 18.,  19.,  20.,  21.,  22.,  23.,  24.,  25.,  26.],
       [ 27.,  28.,  29.,  30.,  31.,  32.,  33.,  34.,  35.],
       [ 36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,  44.]])
```

In all processes, we have:

```
>>> distbuffer = local_array.__distarray__()
>>> distbuffer.keys()
['buffer', 'dim_data', '__version__']
>>> distbuffer['__version__']
'0.10.0'
```

The local arrays, on each separate engine:


Local Arrays
Process (0, 0)
0 1 2 3 4
9 10 11 12 13
18 19 20 21 22
Process (0, 1)
5 6 7 8
14 15 16 17
23 24 25 26
Process (1, 0)
27 28 29 30 31
36 37 38 39 40
Process (1, 1)
32 33 34 35
41 42 43 44

In process (0, 0):

```
>>> distbuffer['buffer']
array([[  0.,   1.,   2.,   3.,   4.],
       [  9.,  10.,  11.,  12.,  13.],
       [ 18.,  19.,  20.,  21.,  22.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5,
  'start': 0,
  'stop': 3},
 {'dist_type': 'b',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9,
  'start': 0,
  'stop': 5})
```

In process (0, 1):

```
>>> distbuffer['buffer']
array([[  5.,   6.,   7.,   8.],
       [ 14.,  15.,  16.,  17.],
       [ 23.,  24.,  25.,  26.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5,
  'start': 0,
  'stop': 3},
 {'dist_type': 'b',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9,
  'start': 5,
  'stop': 9})
```

In process (1, 0):

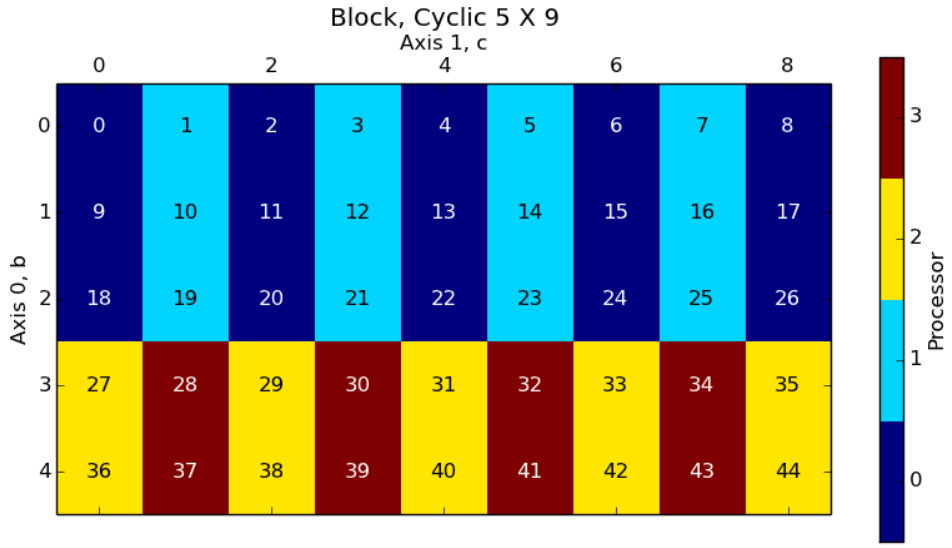
```
>>> distbuffer['buffer']
array([[ 27.,  28.,  29.,  30.,  31.],
       [ 36.,  37.,  38.,  39.,  40.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5,
  'start': 3,
  'stop': 5},
 {'dist_type': 'b',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9,
  'start': 0,
  'stop': 5})
```

In process (1, 1):

```
>>> distbuffer['buffer']
array([[ 32.,  33.,  34.,  35.],
       [ 41.,  42.,  43.,  44.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5,
  'start': 3,
  'stop': 5},
 {'dist_type': 'b',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9,
  'start': 5,
  'stop': 9})
```

2.7 Block, Cyclic

A (5 X 9) array, with a Block, Cyclic (‘b’ X ‘c’) distribution over a (2 X 2) process grid.



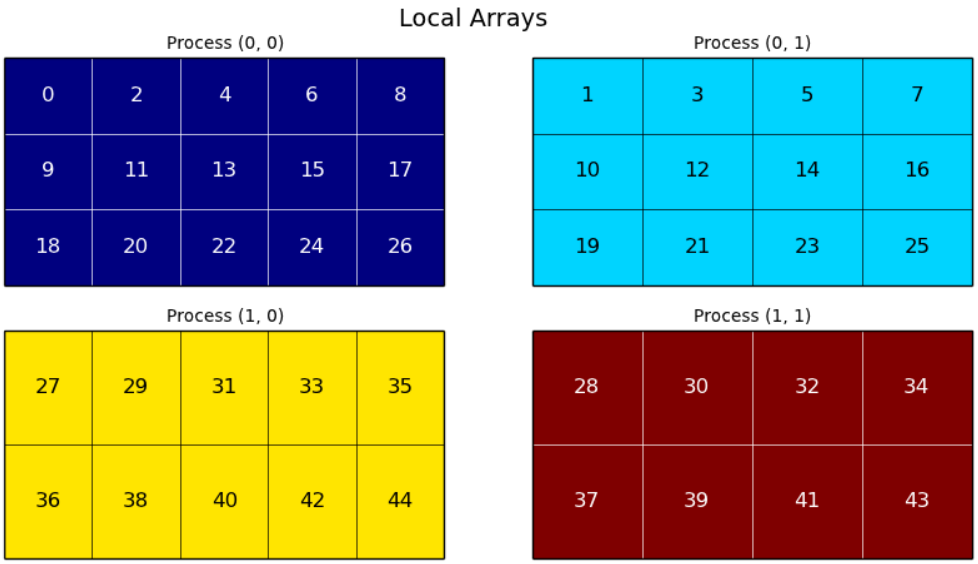
The full (undistributed) array:

```
>>> full_array
array([[  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.],
       [ 18.,  19.,  20.,  21.,  22.,  23.,  24.,  25.,  26.],
       [ 27.,  28.,  29.,  30.,  31.,  32.,  33.,  34.,  35.],
       [ 36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,  44.]])
```

In all processes, we have:

```
>>> distbuffer = local_array.__distarray__()
>>> distbuffer.keys()
['buffer', 'dim_data', '__version__']
>>> distbuffer['__version__']
'0.10.0'
```

The local arrays, on each separate engine:


Local Arrays
Process (0, 0)
0 2 4 6 8
9 11 13 15 17
18 20 22 24 26
Process (0, 1)
1 3 5 7
10 12 14 16
19 21 23 25
Process (1, 0)
27 29 31 33 35
36 38 40 42 44
Process (1, 1)
28 30 32 34
37 39 41 43

In process (0, 0):

```
>>> distbuffer['buffer']
array([[  0.,   2.,   4.,   6.,   8.],
       [  9.,  11.,  13.,  15.,  17.],
       [ 18.,  20.,  22.,  24.,  26.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5,
  'start': 0,
  'stop': 3},
 {'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9,
  'start': 0})
```

In process (0, 1):

```
>>> distbuffer['buffer']
array([[  1.,   3.,   5.,   7.],
       [ 10.,  12.,  14.,  16.],
       [ 19.,  21.,  23.,  25.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5,
  'start': 0,
  'stop': 3},
 {'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9,
  'start': 1})
```

In process (1, 0):

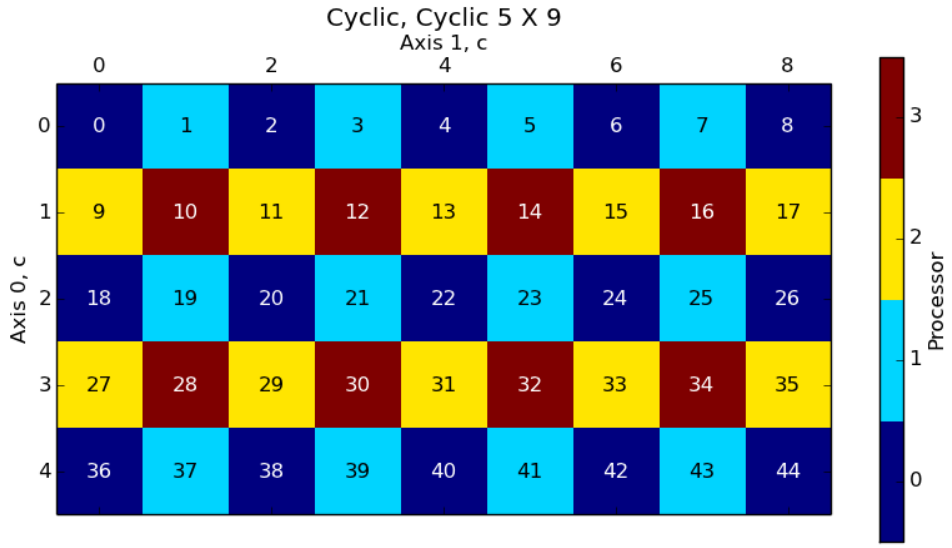
```
>>> distbuffer['buffer']
array([[ 27.,  29.,  31.,  33.,  35.],
       [ 36.,  38.,  40.,  42.,  44.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5,
  'start': 3,
  'stop': 5},
 {'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9,
  'start': 0})
```

In process (1, 1):

```
>>> distbuffer['buffer']
array([[ 28.,  30.,  32.,  34.],
       [ 37.,  39.,  41.,  43.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5,
  'start': 3,
  'stop': 5},
 {'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9,
  'start': 1})
```

2.8 Cyclic, Cyclic

A (5 X 9) array, with a Cyclic, Cyclic ('c' X 'c') distribution over a (2 X 2) process grid.



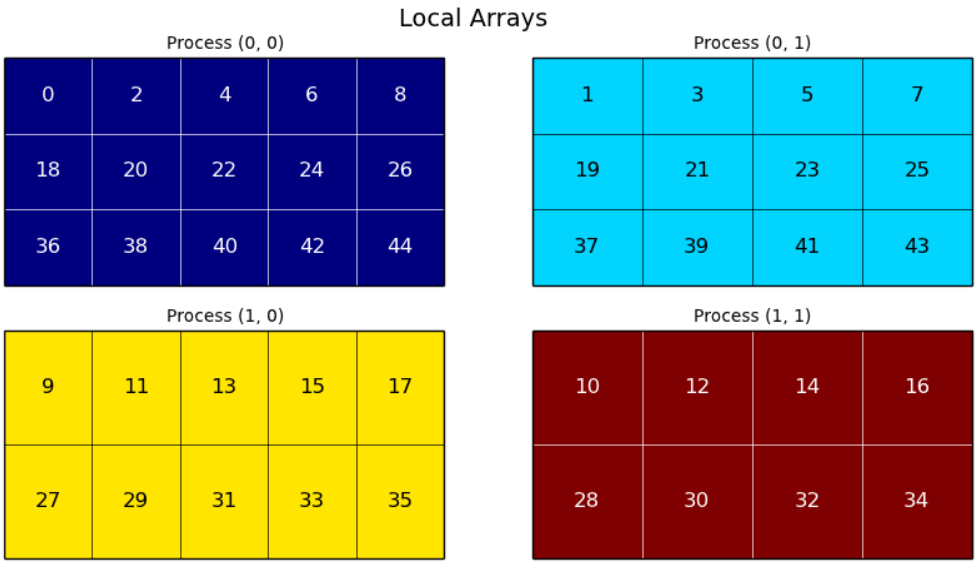
The full (undistributed) array:

```
>>> full_array
array([[  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.],
       [ 18.,  19.,  20.,  21.,  22.,  23.,  24.,  25.,  26.],
       [ 27.,  28.,  29.,  30.,  31.,  32.,  33.,  34.,  35.],
       [ 36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,  44.]])
```

In all processes, we have:

```
>>> distbuffer = local_array.__distarray__()
>>> distbuffer.keys()
['buffer', 'dim_data', '__version__']
>>> distbuffer['__version__']
'0.10.0'
```

The local arrays, on each separate engine:



In process (0, 0):

```
>>> distbuffer['buffer']
array([[  0.,   2.,   4.,   6.,   8.],
       [ 18.,  20.,  22.,  24.,  26.],
       [ 36.,  38.,  40.,  42.,  44.]])
>>> distbuffer['dim_data']
({'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5,
  'start': 0},
 {'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9,
  'start': 0})
```

In process (0, 1):

```
>>> distbuffer['buffer']
array([[  1.,   3.,   5.,   7.],
       [ 19.,  21.,  23.,  25.],
       [ 37.,  39.,  41.,  43.]])
>>> distbuffer['dim_data']
({'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5,
  'start': 0},
 {'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9,
  'start': 1})
```

In process (1, 0):

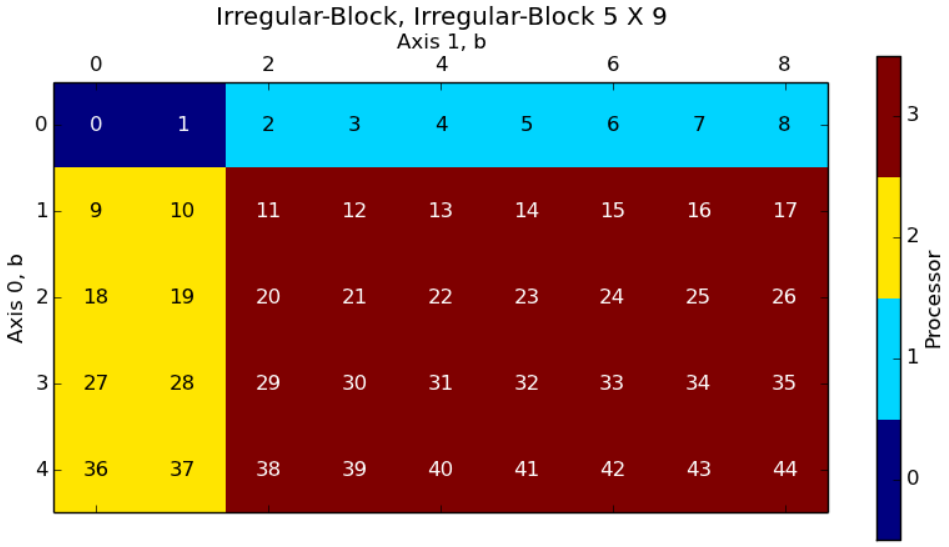
```
>>> distbuffer['buffer']
array([[  9.,  11.,  13.,  15.,  17.],
       [ 27.,  29.,  31.,  33.,  35.]])
>>> distbuffer['dim_data']
({'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5,
  'start': 1},
 {'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9,
  'start': 0})
```

In process (1, 1):

```
>>> distbuffer['buffer']
array([[ 10.,  12.,  14.,  16.],
       [ 28.,  30.,  32.,  34.]])
>>> distbuffer['dim_data']
({'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5,
  'start': 1},
 {'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9,
  'start': 1})
```

2.9 Irregular-Block, Irregular-Block

A (5 X 9) array, with an Irregular-Block, Irregular-Block ('b' X 'b') distribution over a (2 X 2) process grid.



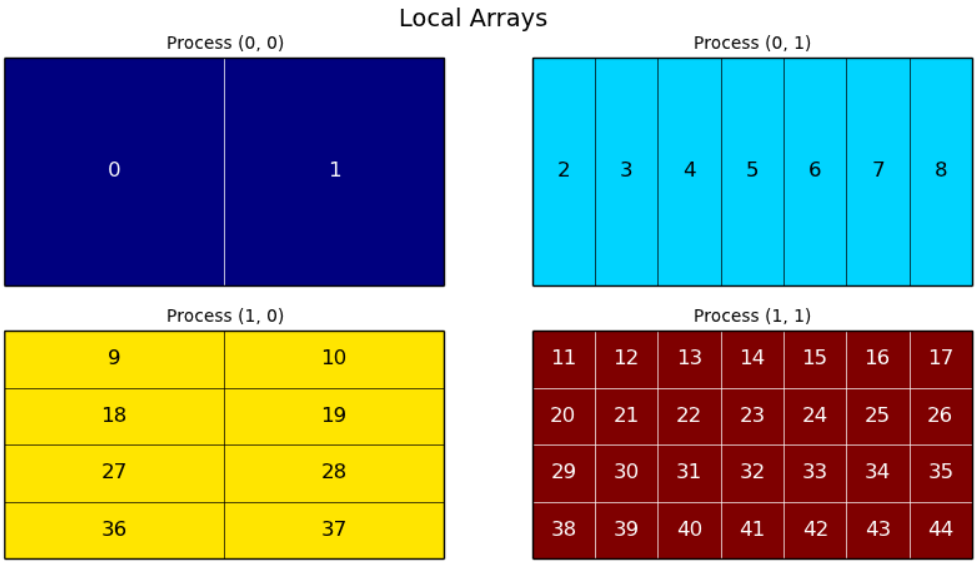
The full (undistributed) array:

```
>>> full_array
array([[  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.],
       [ 18.,  19.,  20.,  21.,  22.,  23.,  24.,  25.,  26.],
       [ 27.,  28.,  29.,  30.,  31.,  32.,  33.,  34.,  35.],
       [ 36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,  44.]])
```

In all processes, we have:

```
>>> distbuffer = local_array.__distarray__()
>>> distbuffer.keys()
['buffer', 'dim_data', '__version__']
>>> distbuffer['__version__']
'0.10.0'
```

The local arrays, on each separate engine:


Local Arrays
Process (0, 0)
0
1
Process (0, 1)
2
3
4
5
6
7
8
Process (1, 0)
9
10
18
19
27
28
36
37
Process (1, 1)
11
12
13
14
15
16
17
20
21
22
23
24
25
26
29
30
31
32
33
34
35
38
39
40
41
42
43
44

In process (0, 0):

```
>>> distbuffer['buffer']
array([[ 0.,  1.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5,
  'start': 0,
  'stop': 1},
 {'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9,
  'start': 0,
  'stop': 2})
```

In process (0, 1):

```
>>> distbuffer['buffer']
array([[ 2.,  3.,  4.,  5.,  6.,  7.,  8.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5,
  'start': 0,
  'stop': 1},
 {'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9,
  'start': 2,
  'stop': 9})
```

In process (1, 0):

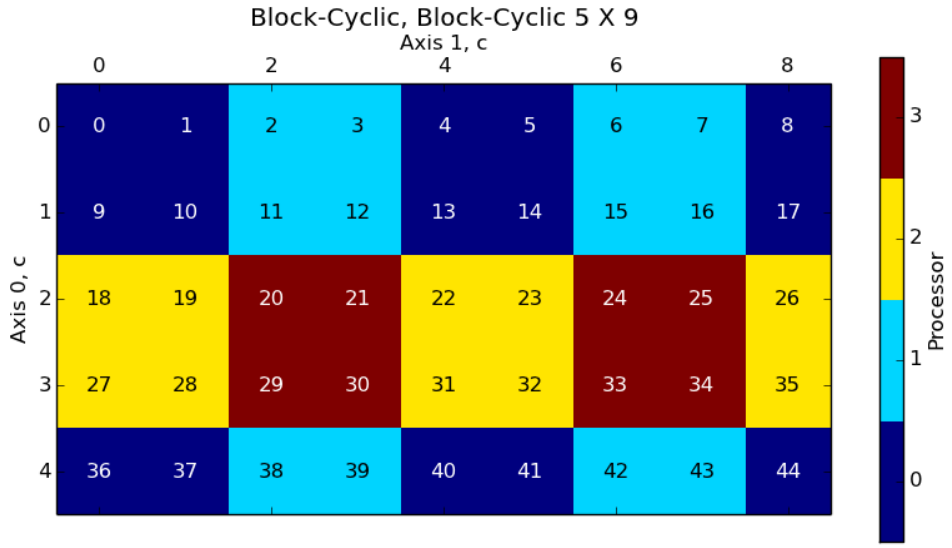
```
>>> distbuffer['buffer']
array([[  9.,  10.],
       [ 18.,  19.],
       [ 27.,  28.],
       [ 36.,  37.]])
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5,
  'start': 1,
  'stop': 5},
 {'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9,
  'start': 0,
  'stop': 2})
```

In process (1, 1):

```
>>> distbuffer['buffer']
array([[ 11.,  12.,  13.,  14.,  15.,  16.,  17.],
       [ 20.,  21.,  22.,  23.,  24.,  25.,  26.],
       [ 29.,  30.,  31.,  32.,  33.,  34.,  35.],
       [ 38.,  39.,  40.,  41.,  42.,  43.,  44.]]
>>> distbuffer['dim_data']
({'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5,
  'start': 1,
  'stop': 5},
 {'dist_type': 'b',
  'padding': [0, 0],
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9,
  'start': 2,
  'stop': 9})
```

2.10 Block-Cyclic, Block-Cyclic

A (5 X 9) array, with a Block-Cyclic, Block-Cyclic ('c' X 'c') distribution over a (2 X 2) process grid.



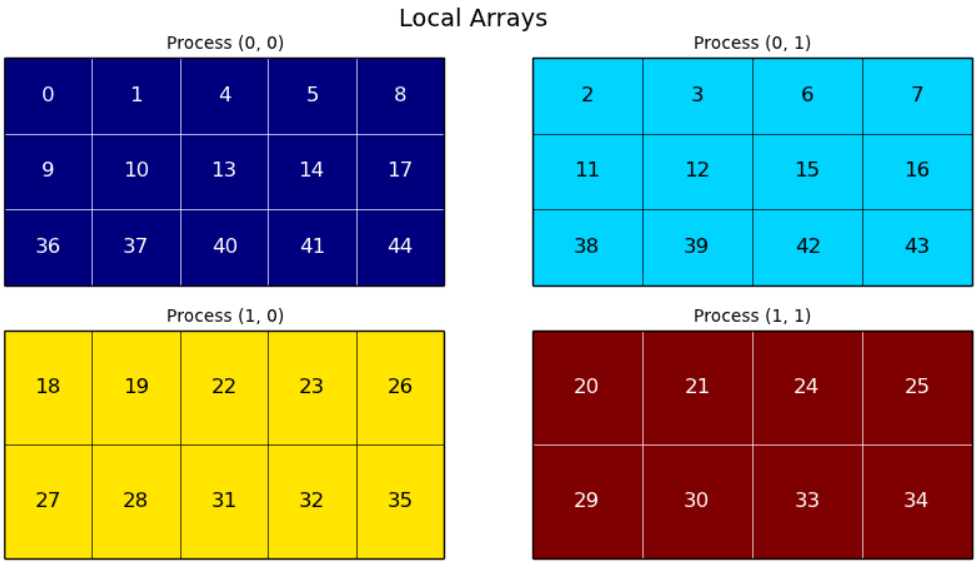
The full (undistributed) array:

```
>>> full_array
array([[  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.],
       [ 18.,  19.,  20.,  21.,  22.,  23.,  24.,  25.,  26.],
       [ 27.,  28.,  29.,  30.,  31.,  32.,  33.,  34.,  35.],
       [ 36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,  44.]])
```

In all processes, we have:

```
>>> distbuffer = local_array.__distarray__()
>>> distbuffer.keys()
['buffer', 'dim_data', '__version__']
>>> distbuffer['__version__']
'0.10.0'
```

The local arrays, on each separate engine:


Local Arrays
Process (0, 0)
0 1 4 5 8
9 10 13 14 17
36 37 40 41 44
Process (0, 1)
2 3 6 7
11 12 15 16
38 39 42 43
Process (1, 0)
18 19 22 23 26
27 28 31 32 35
Process (1, 1)
20 21 24 25
29 30 33 34

In process (0, 0):

```
>>> distbuffer['buffer']
array([[  0.,   1.,   4.,   5.,   8.],
       [  9.,  10.,  13.,  14.,  17.],
       [ 36.,  37.,  40.,  41.,  44.]])
>>> distbuffer['dim_data']
({'block_size': 2,
  'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5,
  'start': 0},
 {'block_size': 2,
  'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9,
  'start': 0})
```

In process (0, 1):

```
>>> distbuffer['buffer']
array([[  2.,   3.,   6.,   7.],
       [ 11.,  12.,  15.,  16.],
       [ 38.,  39.,  42.,  43.]])
>>> distbuffer['dim_data']
({'block_size': 2,
  'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5,
  'start': 0},
 {'block_size': 2,
  'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9,
  'start': 2})
```

In process (1, 0):

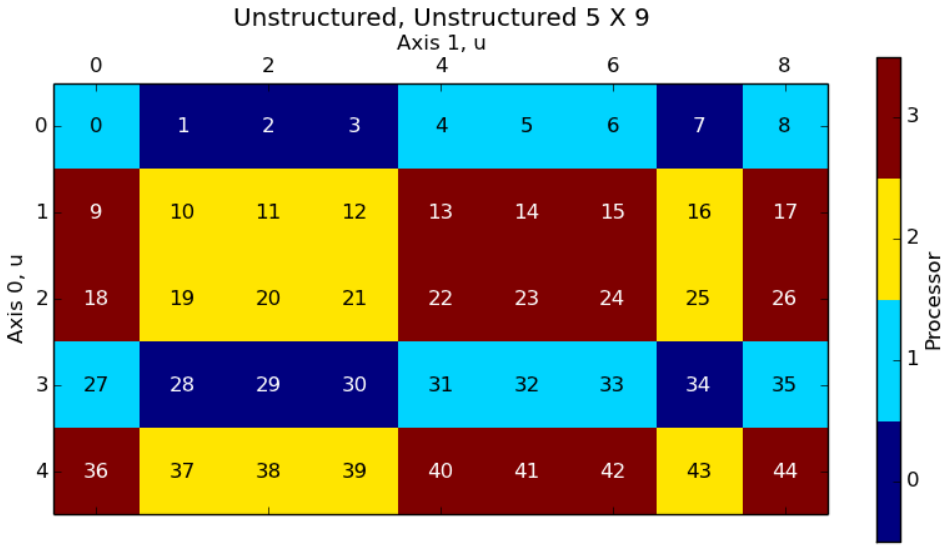
```
>>> distbuffer['buffer']
array([[ 18.,  19.,  22.,  23.,  26.],
       [ 27.,  28.,  31.,  32.,  35.]])
>>> distbuffer['dim_data']
({'block_size': 2,
  'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5,
  'start': 2},
 {'block_size': 2,
  'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9,
  'start': 0})
```

In process (1, 1):

```
>>> distbuffer['buffer']
array([[ 20.,  21.,  24.,  25.],
       [ 29.,  30.,  33.,  34.]])
>>> distbuffer['dim_data']
({'block_size': 2,
  'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5,
  'start': 2},
 {'block_size': 2,
  'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9,
  'start': 2})
```

2.11 Unstructured, Unstructured

A (5 X 9) array, with an Unstructured, Unstructured ('u' X 'u') distribution over a (2 X 2) process grid.



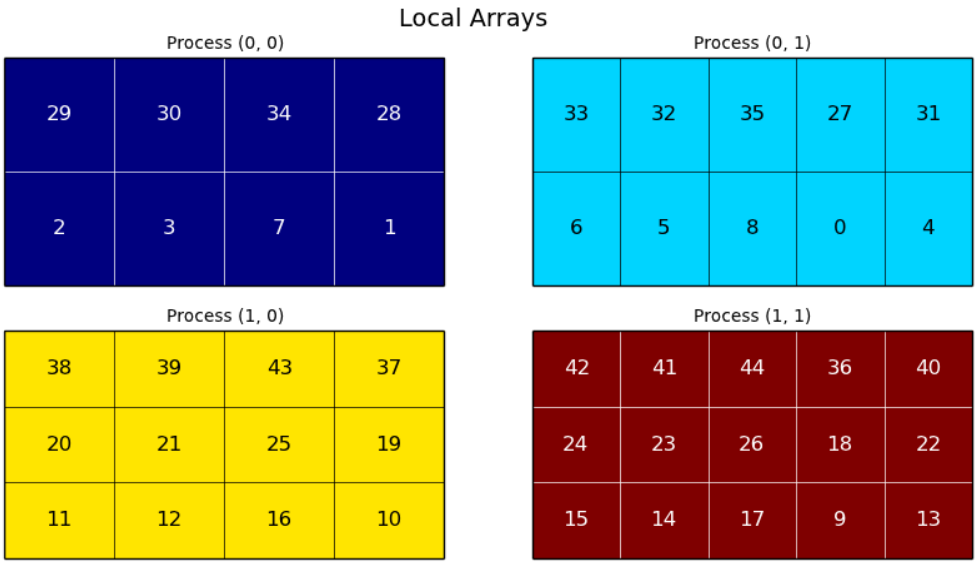
The full (undistributed) array:

```
>>> full_array
array([[  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.],
       [ 18.,  19.,  20.,  21.,  22.,  23.,  24.,  25.,  26.],
       [ 27.,  28.,  29.,  30.,  31.,  32.,  33.,  34.,  35.],
       [ 36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,  44.]])
```

In all processes, we have:

```
>>> distbuffer = local_array.__distarray__()
>>> distbuffer.keys()
['buffer', 'dim_data', '__version__']
>>> distbuffer['__version__']
'0.10.0'
```

The local arrays, on each separate engine:



In process (0, 0):

```
>>> distbuffer['buffer']
array([[ 29.,  30.,  34.,  28.],
       [  2.,   3.,   7.,   1.]])
>>> distbuffer['dim_data']
({'dist_type': 'u',
  'indices': array([3, 0]),
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5},
 {'dist_type': 'u',
  'indices': array([2, 3, 7, 1]),
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9})
```

In process (0, 1):

```
>>> distbuffer['buffer']
array([[ 33.,  32.,  35.,  27.,  31.],
       [  6.,   5.,   8.,   0.,   4.]])
>>> distbuffer['dim_data']
({'dist_type': 'u',
  'indices': array([3, 0]),
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5},
 {'dist_type': 'u',
  'indices': array([6, 5, 8, 0, 4]),
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9})
```

In process (1, 0):

```
>>> distbuffer['buffer']
array([[ 38.,  39.,  43.,  37.],
       [ 20.,  21.,  25.,  19.],
       [ 11.,  12.,  16.,  10.]])
>>> distbuffer['dim_data']
({'dist_type': 'u',
  'indices': array([4, 2, 1]),
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5},
 {'dist_type': 'u',
  'indices': array([2, 3, 7, 1]),
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9})
```

In process (1, 1):

```
>>> distbuffer['buffer']
array([[ 42.,  41.,  44.,  36.,  40.],
       [ 24.,  23.,  26.,  18.,  22.],
       [ 15.,  14.,  17.,   9.,  13.]])
>>> distbuffer['dim_data']
({'dist_type': 'u',
  'indices': array([4, 2, 1]),
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5},
 {'dist_type': 'u',
  'indices': array([6, 5, 8, 0, 4]),
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9})
```

2.12 Cyclic, Block, Cyclic

A (5 X 9 X 3) array, with a Cyclic, Block, Cyclic ('c' X 'b' X 'c') distribution over a (2 X 2 X 2) process grid.

The full (undistributed) array:

```
>>> full_array
array([[[   0.,    1.,    2.],
        [   3.,    4.,    5.],
        [   6.,    7.,    8.],
        [   9.,   10.,   11.],
        [  12.,   13.,   14.],
        [  15.,   16.,   17.],
        [  18.,   19.,   20.],
        [  21.,   22.,   23.],
        [  24.,   25.,   26.]],
       [[  27.,   28.,   29.],
        [  30.,   31.,   32.],
        [  33.,   34.,   35.],
        [  36.,   37.,   38.],
        [  39.,   40.,   41.],
        [  42.,   43.,   44.],
        [  45.,   46.,   47.],
        [  48.,   49.,   50.],
        [  51.,   52.,   53.]],
       [[  54.,   55.,   56.],
        [  57.,   58.,   59.],
        [  60.,   61.,   62.],
        [  63.,   64.,   65.],
        [  66.,   67.,   68.],
        [  69.,   70.,   71.],
        [  72.,   73.,   74.],
        [  75.,   76.,   77.],
        [  78.,   79.,   80.]],
       [[  81.,   82.,   83.],
        [  84.,   85.,   86.],
        [  87.,   88.,   89.],
        [  90.,   91.,   92.],
        [  93.,   94.,   95.],
        [  96.,   97.,   98.],
        [  99.,  100.,  101.],
        [ 102.,  103.,  104.],
        [ 105.,  106.,  107.]],
       [[ 108.,  109.,  110.],
        [ 111.,  112.,  113.],
        [ 114.,  115.,  116.],
        [ 117.,  118.,  119.],
        [ 120.,  121.,  122.],
        [ 123.,  124.,  125.],
        [ 126.,  127.,  128.],
        [ 129.,  130.,  131.],
        [ 132.,  133.,  134.]]])
```

In all processes, we have:

```
>>> distbuffer = local_array.__distarray__()
>>> distbuffer.keys()
['buffer', 'dim_data', '__version__']
>>> distbuffer['__version__']
```

```
'0.10.0'
```

The local arrays, on each separate engine:

In process (0, 0, 0):

```
>>> distbuffer['buffer']
array([[[   0.,    2.],
        [   3.,    5.],
        [   6.,    8.],
        [   9.,   11.],
        [  12.,   14.]],
       [[  54.,   56.],
        [  57.,   59.],
        [  60.,   62.],
        [  63.,   65.],
        [  66.,   68.]],
       [[ 108.,  110.],
        [ 111.,  113.],
        [ 114.,  116.],
        [ 117.,  119.],
        [ 120.,  122.]]])
>>> distbuffer['dim_data']
({'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5,
  'start': 0},
 {'dist_type': 'b',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9,
  'start': 0,
  'stop': 5},
 {'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 3,
  'start': 0})
```

In process (0, 0, 1):

```
>>> distbuffer['buffer']
array([[[   1.],
        [   4.],
        [   7.],
        [  10.],
        [  13.]],
       [[  55.],
        [  58.],
        [  61.],
        [  64.],
        [  67.]],
       [[ 109.],
        [ 112.],
        [ 115.],
        [ 118.],
        [ 121.]]])
>>> distbuffer['dim_data']
 ({'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5,
  'start': 0},
 {'dist_type': 'b',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9,
  'start': 0,
  'stop': 5},
 {'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 3,
  'start': 1})
```

In process (0, 1, 0):

```
>>> distbuffer['buffer']
array([[[  15.,   17.],
        [  18.,   20.],
        [  21.,   23.],
        [  24.,   26.]],
       [[  69.,   71.],
        [  72.,   74.],
        [  75.,   77.],
        [  78.,   80.]],
       [[ 123.,  125.],
        [ 126.,  128.],
        [ 129.,  131.],
        [ 132.,  134.]]])
>>> distbuffer['dim_data']
({'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5,
  'start': 0},
 {'dist_type': 'b',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9,
  'start': 5,
  'stop': 9},
 {'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 3,
  'start': 0})
```

In process (0, 1, 1):

```
>>> distbuffer['buffer']
array([[[  16.],
        [  19.],
        [  22.],
        [  25.]],
       [[  70.],
        [  73.],
        [  76.],
        [  79.]],
       [[ 124.],
        [ 127.],
        [ 130.],
        [ 133.]]])
>>> distbuffer['dim_data']
({'dist_type': 'c',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 5,
  'start': 0},
 {'dist_type': 'b',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9,
  'start': 5,
  'stop': 9},
 {'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 3,
  'start': 1})
```

In process (1, 0, 0):

```
>>> distbuffer['buffer']
array([[[ 27.,  29.],
        [ 30.,  32.],
        [ 33.,  35.],
        [ 36.,  38.],
        [ 39.,  41.]],
       [[ 81.,  83.],
        [ 84.,  86.],
        [ 87.,  89.],
        [ 90.,  92.],
        [ 93.,  95.]]])
>>> distbuffer['dim_data']
({'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5,
  'start': 1},
 {'dist_type': 'b',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9,
  'start': 0,
  'stop': 5},
 {'dist_type': 'c',
```

In process (1, 0, 1):

```
>>> distbuffer['buffer']
array([[[ 28.],
        [ 31.],
        [ 34.],
        [ 37.],
        [ 40.]],
       [[ 82.],
        [ 85.],
        [ 88.],
        [ 91.],
        [ 94.]]])
>>> distbuffer['dim_data']
 ({'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5,
  'start': 1},
 {'dist_type': 'b',
  'proc_grid_rank': 0,
  'proc_grid_size': 2,
  'size': 9,
  'start': 0,
  'stop': 5},
 {'dist_type': 'c',
```

In process (1, 1, 0):

```
>>> distbuffer['buffer']
array([[[  42.,   44.],
        [  45.,   47.],
        [  48.,   50.],
        [  51.,   53.]],
       [[  96.,   98.],
        [  99.,  101.],
        [ 102.,  104.],
        [ 105.,  107.]]])
>>> distbuffer['dim_data']
 ({'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5,
  'start': 1},
 {'dist_type': 'b',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9,
  'start': 5,
  'stop': 9},
 {'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
```

In process (1, 1, 1):

```
>>> distbuffer['buffer']
array([[[  43.],
        [  46.],
        [  49.],
        [  52.]],
       [[  97.],
        [ 100.],
        [ 103.],
        [ 106.]]])
>>> distbuffer['dim_data']
 ({'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 5,
  'start': 1},
 {'dist_type': 'b',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
  'size': 9,
  'start': 5,
  'stop': 9},
 {'dist_type': 'c',
  'proc_grid_rank': 1,
  'proc_grid_size': 2,
```

CHAPTER 3

Appendix

3.1 Computing the number of indices owned by a rank

```
# encoding: utf-8
# ----------------------------------------------------------------------------
#  Copyright (C) 2008-2014, Enthought, Inc.
#  Distributed under the terms of the BSD License.  See COPYING.rst.
# ----------------------------------------------------------------------------

"""
Utility functions described in the Distributed Array Protocol.
"""


def num_owned_indices_from_block(dim_dict):
    """Given a dimension dictionary with dist_type 'b', return the number of
    indices owned, taking padding into account.
    """
    count = dim_dict['stop'] - dim_dict['start']
    padding = dim_dict.get('padding', (0,0))
    left_process = 0
    right_process = dim_dict['proc_grid_size'] - 1

    # Communication padding doesn't count.
    if dim_dict['proc_grid_rank'] != left_process:
        # We're not at the left boundary;
        # padding[0] is communication padding.
        count -= padding[0]
    if dim_dict['proc_grid_rank'] != right_process:
        # We're not at the right boundary;
        # padding[1] is communication padding.
        count -= padding[1]

    return count


def num_owned_indices_from_cyclic(dd):
    """Given a dimension dictionary `dd` with dist_type 'c', return the number
    of indices owned.
    """
    block_size = dd.get('block_size', 1)
    global_nblocks, partial = divmod(dd['size'], block_size)
```

```
    local_nblocks = ((global_nblocks - 1 - dd['proc_grid_rank']) //
                     dd['proc_grid_size']) + 1
    local_partial = partial if dd['proc_grid_rank'] == 0 else 0
    local_size = local_nblocks * dd['block_size'] + local_partial
    return local_size


def num_owned_indices_from_unstructured(dd):
    """Given a dimension dictionary `dd` with dist_type 'u', return the number
    of indices owned.
    """
    indices_buffer = memoryview(dd['indices'])
    return len(indices_buffer)


def num_owned_indices(dd):
    """Given a dimension dictionary `dd` with dist_type 'b', return the number
    of indices owned.
    """
    dist_type = dd['dist_type']
    selector = {'b': num_owned_indices_from_block,
                'c': num_owned_indices_from_cyclic,
                'u': num_owned_indices_from_unstructured,
               }
    return selector[dist_type](dd)
```