Distributed Array Protocol Documentation

Release 0.9.0

Jeff Daily, Brian Granger, Robert Grant, Min Ragan-Kelley, Kurt Sm

February 11, 2014

1 Overview 3

2 Usecases 5

3 Sources 7

4 Definitions 9

5 Exporting a Distributed Array 11

6 Dimension Dictionaries 13
- 6.1 Required key-value pairs 13
- 6.2 Optional key-value pairs 14
- 6.3 Distribution-type specific key-value pairs 14

7 Examples 17
- 7.1 Block, Undistributed 17
- 7.2 Block with padding 18
- 7.3 Unstructured 19

8 References 21

Contents

- Distributed Array Protocol v0.9.0
  - Overview
  - Usecases
  - Sources
  - Definitions
  - Exporting a Distributed Array
  - Dimension Dictionaries
    - Required key-value pairs
    - Optional key-value pairs
    - Distribution-type specific key-value pairs
  - Examples
    - Block, Undistributed
    - Block with padding
    - Unstructured
  - References

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

CHAPTER 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.

Sources

The primary sources and inspiration for the DAP are:

- NumPy [1] and the Revised Buffer Protocol [4]
- Trilinos [2] and PyTrilinos [3]
- Global Arrays [4] and Global Arrays in NumPy (GAiN) [5]
- The Chapel [6], X10 [7], and HP-Fortran [8] languages
- Distributed array implementation in the Julia [9] language

[1] NumPy. http://www.numpy.org/
[2] Trilinos. http://trilinos.sandia.gov/
[3] PyTrilinos. http://trilinos.sandia.gov/packages/pytrilinos/
[4] Global Arrays. http://hpc.pnl.gov/globalarrays/
[5] Global Arrays in NumPy. http://www.pnnl.gov/science/highlights/highlight.asp?id=1043
[6] Chapel. http://chapel.cray.com/
[7] X10. http://x10-lang.org/
[8] High Perfomance Fortran. http://dacnet.rice.edu/
[9] Julia. http://docs.julialang.org

Definitions

process A "process" is the basic unit of execution and is as defined in MPI [1]. It is also analogous to an IPython.parallel [2] engine. 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.

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 boundary padding elements would exist even if the array were not distributed. These elements are included in a distributed dimension's `'size'`.

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] Message Passing Interface. http://www.open-mpi.org/
[2] IPython Parallel. http://ipython.org/ipython-doc/dev/parallel/

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 [1] and PEP-440 [2]. 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, with the zeroth dictionary associated with the zeroth dimension of the array, and so on for each dimension in succession. There is one dimension dictionary per dimension, **whether or not that dimension is distributed**. These dictionaries are intended to include all metadata required to fully specify a distributed array's distribution information.

[1] Semantic Versioning 2.0.0. http://semver.org/

[2] PEP 440: Version Identification and Dependency Specification. http://www.python.org/dev/peps/pep-0440/

CHAPTER 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 (or no distribution for value `'n'`).

The following dist_types are currently supported:

name	dist_type	required keys
undistributed	'n'	'dist_type', 'size'
block	'b'	common, 'start', 'stop'
cyclic	'c'	common, 'start'
unstructured	'u'	common, 'indices'

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

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

6.1 Required key-value pairs

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

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

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

- `'size'` : `int`, >= 0

  Total number of global array elements along this dimension.

All *distributed* dimensions shall have the following keys in their dimension dictionary, with the associated value:

- `'proc_grid_size'` : `int`, >= 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_sizes for all distributed dimensions shall equal the total number of processes in the communicator.

- `proc_grid_rank` : `int`

  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 MPI standard guarantees that Cartesian process coordinates are always assigned to ranks in the same way [1].

6.2 Optional key-value pairs

- `'periodic'` : `bool`

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

6.3 Distribution-type specific key-value pairs

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

- undistributed (`dist_type` is `'n'`):
  - `padding` : optional. see same key under block distribution.
- 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

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

    For a block-distributed dimension, adjacent processes as determined by the dimension dictionary's `proc_grid_rank` field shall have adjacent global index ranges, i.e., for two processes `a` and `b` with grid ranks `i` and `i+1` respectively, the `stop` of `a` shall be equal to the `start` of `b`. Processes may contain differently-sized global index ranges.
  - `padding` : 2-tuple of `int`, each greater than or equal to zero. Optional.

    When present, indicates the number of "padding" values at the lower and upper limits (respectively) of the indices available on this process. This padding can be either "boundary padding" or "communication padding". When not present, indicates that the distributed array is not padded in this dimension on any process.

    Whenever an element of the `padding` tuple is > 0 and the padding is on an internal edge of the process grid (or the dimension is periodic), that indicates this is "communication padding", and the communication padding elements do not count towards the `size` of the array in this dimension. In other words, the array shares the indicated number of indices with its neighbor (as determined by `proc_grid_rank`), and further, this neighboring process owns the data. When an element of the `padding` tuple is > 0 and the padding is on an external edge of the process grid (and the dimension is not periodic), that indicates that this is "boundary padding".

    Padding is an all-or-nothing attribute: if the `padding` keyword is present in any dimension dictionary for a dimension of the distributed array, then the `padding` keyword shall be present on *all* processes for the same dimension dictionary. The value associated with `padding` can be the tuple `(0,0)` indicating that this local array is not padded in this dimension, but other local arrays may be padded in this dimension.
- cyclic (`dist_type` is `'c'`):

[1] MPI-2.2 Standard: Virtual Topologies. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node165.htm#Node165

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

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

    The cyclic distribution is what results from assigning global indices–or contiguous blocks of indices, in the case when `block_size` is greater than one–to processes in round-robin fashion. When `block_size` equals one, a constraint for cyclic is that the Python slice formed from the `start`, `size`, and `proc_grid_size` values reproduces the local array's indices as in standard NumPy slicing.

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

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

    If `block_size == 1`, then this is the "true" cyclic distribution as specified by ScaLAPACK [2]; if `1 < block_size < size // proc_grid_size`, then this dist type specifies the block-cyclic distribution [16] [3]. Block-cyclic can be thought of as analogous to the cyclic distribution, but it distributes contiguous blocks of global indices in round robin fashion rather than single indices. In this way block-cyclic is a generalization of the block and cyclic distribution types (for an evenly distributed block distribution). When `block_size == ceil(size / proc_grid_size)`, block cyclic is equivalent to block.

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

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

    Global indices available on this process.

    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.

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

[3] 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

Examples

7.1 Block, Undistributed

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-distributed 0th dimension and an undistributed 1st dimension.

In process 0:

```
>>> distbuffer = a0.__distarray__()
>>> distbuffer.keys()
['__version__', 'buffer', 'dim_data']
>>> distbuffer['__version__']
'0.9.0'
>>> 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': 'n'})
```

In process 1:

```
>>> distbuffer = a1.__distarray__()
>>> distbuffer.keys()
['__version__', 'buffer', 'dim_data']
>>> distbuffer['__version__']
'0.9.0'
>>> 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': 'n'})
```

7.2 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 the communication padding is not counted toward the size (though the boundary padding is).

For this example, the global index arrangement on each processor, with 'B' for boundary and 'C' for communication elements, are arranged 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__']
'0.9.0'
>>> 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': 9,
  'padding': (1, 1)})
```

In process 1:

```
>>> distbuffer = a1.__distarray__()
>>> distbuffer.keys()
['__version__', 'buffer', 'dim_data']
>>> distbuffer['__version__']
'0.9.0'
>>> 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': 9,
  'stop': 18,
  'padding': (1, 1)})
```

7.3 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__']
'0.9.0'
>>> 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': [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': [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': [10, 25,  5, 21,  7, 18, 11, 26, 29, 24, 23, 28, 14,
              20,  9, 16, 27,  8, 17, 22]},)
```

CHAPTER 8

References