


METHODOLOGY

Open Access



D2O: a distributed data object for parallel high-performance computing in Python

Theo Steininger^{1,2*} , Maksim Greiner^{1,2}, Frederik Beaujean^{2,3} and Torsten Enßlin^{1,2}

*Correspondence:
theos@mpa-garching.mpg.de
¹ Max Planck Institut
für Astrophysik,
Karl-Schwarzschild-Strasse 1,
85741 Garching, Germany
Full list of author information
is available at the end of the
article

Abstract

We introduce `D2O`, a Python module for cluster-distributed multi-dimensional numerical arrays. It acts as a layer of abstraction between the algorithm code and the data-distribution logic. The main goal is to achieve usability without losing numerical performance and scalability. `D2O`'s global interface is similar to the one of a `numpy.ndarray`, whereas the cluster node's local data is directly accessible for use in customized high-performance modules. `D2O` is written in pure Python which makes it portable and easy to use and modify. Expensive operations are carried out by dedicated external libraries like `numpy` and `mpi4py`. The performance of `D2O` is on a par with `numpy` for serial applications and scales well when moving to an MPI cluster. `D2O` is open-source software available under the GNU General Public License v3 (GPL-3) at <https://gitlab.mpcdf.mpg.de/ift/D2O>.

Keywords: Parallelization, Numerics, MPI, Python, Numpy, Open source

Introduction

Background

Data sets in simulation and signal-reconstruction applications easily reach sizes too large for a single computer's random access memory (RAM). A reasonable grid size for such tasks like galactic density reconstructions [1] or multi-frequency imaging in radio astronomy [2] is a cube with a side resolution of 2048. Such a cube contains $2048^3 \approx 8.6 \cdot 10^9$ voxels. Storing a 64-bit double for every voxel therefore consumes 64 GiB. In practice one has to handle several or even many instances of those arrays which ultimately prohibits the use of single shared memory machines. Apart from merely holding the arrays' data in memory, parallelization is needed to process those huge arrays within reasonable time. This applies to basic arithmetics like addition and multiplication as well as to complex operations like Fourier transformation and advanced linear algebra, e.g. operator inversions or singular value decompositions. Thus parallelization is highly advisable for code projects that must be scaled to high resolutions.

To be specific, the initial purpose of `D2O` was to provide parallelization to the package for Numerical Information Field Theory (NIFTY)[3], which permits the abstract and efficient implementation of sophisticated signal processing methods. Typically, those methods are so complex on their own that a NIFTY user should not need to bother with parallelization details in addition to that. It turned out that providing a generic encapsulation for parallelization to NIFTY is not straightforward as the applications NIFTY is

used for are highly diversified. The challenge hereby is that, despite all their peculiarities, for those applications numerical efficiency is absolutely crucial. Hence, for encapsulating the parallelization effort in NIFTY we needed an approach that is flexible enough to adapt to those different applications such that numerical efficiency can be preserved: D2O.

D2O is implemented in Python. As a high-level language with a low-entry barrier Python is widely used in computational science. It has a huge standard library and an active community for 3rd party packages. For computationally demanding applications Python is predominantly used as a steering language for external compiled modules because Python itself is slow for numerics.

This article is structured as follows. "Aim" section gives the aims of D2O, and "Alternative packages" section describes alternative data distribution packages. We discuss the code architecture in "Code architecture" section, the basic usage of D2O in "Basic usage" section, and the numerical scaling behavior in section "Performance and scalability". "Summary and outlook" section contains our conclusion and "Appendix 1" describes the detailed usage of D2O.

Aim

As most scientists are not fully skilled software engineers, for them the hurdle for developing parallelized code is high. Our goal is to provide data scientists with a numpy array-like object (cf. *numpy* [4]) that distributes data among several nodes of a cluster in a user-controllable way. The user, however, shall not need to have profound knowledge about parallel programming with a system like *MPI* [5, 6] to achieve this. The transition to use *distributed_data_objects* instead of numpy arrays in existing code must be as straightforward as possible. Hence, D2O shall in principle run—at least in a non-parallelized manner—with standard-library dependencies available; the packages needed for parallel usage should be easily available. Whilst providing a global-minded interface, the node's local data should be directly accessible in order to enable the usage in specialized high-performance modules. This approach matches with the theme of *DistArray* [7]: "Think globally, act locally". Regarding D2O's architecture we do not want to make any a-priori assumptions about the specific distribution strategy, but retain flexibility: it shall be possible to adapt to specific constraints induced from third-party libraries a user may incorporate. For example, a library for fast Fourier transformations like *FFTW* [8] may rely on a different data-distribution model than a package for linear algebra operations like *ScaLAPACK* [9].¹ In the same manner it shall not matter whether a new distribution scheme stores data redundantly or not, e.g. when a node is storing not only a distinct piece of a global array, but also its neighboring (ghost) cells [10].

Our main focus is on rendering extremely costly computations possible in the first place; not on improving the speed of simple computations that can be done serially. Although primarily geared towards *weak scaling*, it turns out that D2O performs very well in *strong-scaling* scenarios, too; see "Performance and scalability" section for details.

¹ FFTW distributes slices of data, while ScaLAPACK uses a block-cyclic distribution pattern.

Alternative packages

There are several alternatives to D2O. We discuss the differences to D2O and why the alternatives are not sufficient for our needs.

DistArray

DistArray [7] is very mature and powerful. Its approach is very similar to D2O: It mimics the interface of a multi dimensional numpy array while distributing the data among nodes in a cluster. However, *DistArray* involves a design decision that makes it inapt for our purposes: it has a strict client-worker architecture. *DistArray* either needs an *ipython ipcluster* [11] as back end or must be run with two or more MPI processes. The former must be started before an interactive ipython session is launched. This at least complicates the workflow in the prototyping phase and at most is not practical for batch system based computing on a cluster. The latter enforces tool-developers who build on top of *DistArray* to demand that their code always is run parallelized. Both scenarios conflict with our goal of minimal second order dependencies and maximal flexibility, cf. "Aim" section. Nevertheless, its theme also applies to D2O: "Think globally, act locally".

Scalapy (ScaLAPACK)

scalapy is a Python wrapper around *ScaLAPACK* [9], which is "a library of high-performance linear algebra routines for parallel distributed memory machines" [12]. The `scalapy.DistributedMatrix` class essentially uses the routines from *ScaLAPACK* and therefore is limited to the functionality of that: two-dimensional arrays and very specific block-cyclic distribution strategies that optimize numerical efficiency in the context of linear algebra problems. In contrast, we are interested in n -dimensional arrays whose distribution scheme shall be arbitrary in the first place. Therefore *scalapy* is not extensive enough for us.

Petsc4py (PETSc)

petsc4py is a Python wrapper around *PETSc*, which "is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations" [13]. Regarding distributed arrays its scope is as focused as *scalapy* to its certain problem domain—here: solving partial differential equations. The class for distributed arrays `petsc4py.PETSc.DMDA` is limited to one, two and three dimensions as *PETSc* uses a highly problem-fitted distribution scheme. We in contrast need n -dimensional arrays with arbitrary distribution schemes. Hence, *petsc4py* is not suitable for us.

Code architecture

Choosing the right level of parallelization

D2O distributes numerical arrays over a cluster in order to parallelize and therefore to speed up operations on the arrays themselves. An application that is built on top of D2O can profit from its fast array operations that may be performed on a cluster. However, there are various approaches how to deploy an algorithm on a cluster and D2O implements only one of them. In order to understand the design decisions of D2O and its position respective to other packages, cf. "Alternative packages" section, we will now discuss the general problem setting of parallelization and possible approaches for that. Thereby we reenact the decision process which led to the characteristics D2O has today.

Vertical and horizontal scaling

Suppose we want to solve an expensive numerical problem which involves operations on data arrays. To reduce the computation time one can in principle do two things. Either use a faster machine—*vertical scaling*—or use more than one machine—*horizontal scaling*. Vertical scaling has the advantage that existing code does not need to be changed,² but in many cases this is not appropriate. Maybe one already uses the fastest possible machine, scaling up is not affordable or even the fastest machine available is still too slow. Because of this, we choose horizontal scaling.

High- and low-level parallelization

With horizontal scaling we again face two choices: high- and low-level parallelization. With high-level parallelization, many replicas of the algorithm run simultaneously, potentially on multiple machines. Each instance then works independently if possible, solving an isolated part of the global problem. At the end, the individual results get collected and merged. The python framework *pathos* [14] provides functionality for this kind of procedure.

An example of high-level parallelization is a sample generator which draws from a probability distribution. Using high-level parallelization many instances of the generator produce their own samples, which involves very little communication overhead. The sample production process itself, however, is not sped up.

In low-level parallelization, several nodes work together on one basic task at a time. For the above sample generator, this means that all nodes work on the same sample at a time. Hence, the time needed for producing individual samples is reduced; they are serially generated by the cluster as a whole.

Downsides

Both of these approaches have their drawbacks. For high-level parallelization the algorithm itself must be parallelizable. Every finite algorithm has a maximum degree of intrinsic parallelization.³ If this degree is lower than the desired number of processes then high-level parallelization reaches its limits. This is particularly true for algorithms that cannot be parallelized by themselves, like iterative schemes. Furthermore, there can be an additional complication: if the numerical problem deals with extremely large objects it may be the case that it is not at all solvable by one machine alone.⁴

Now let us consider low-level parallelization. As stated above, we assume that the solution of the given numerical problem involves operations on data arrays. Examples for those are unary,⁵ binary⁶ or sorting operations, but also more advanced procedures like Fourier transformations or (other) linear algebra operations. Theoretically, the absolute maximum degree of intrinsic parallelization for an array operation is equal to the array's number of elements. For comparison, the problems we want to tackle involve at least 10^8 elements but most of the TOP500 [15] supercomputers possess 10^6 cores or

² This is true if scaling up does not involve a change of the processor architecture.

³ For the exemplary sample generator the maximum degree of parallelization is the total number of requested samples.

⁴ In case of the sample generator this would be the case if even one sample would be too large for an individual machine's RAM.

⁵ E.g. the positive, negative or absolute values of the array's individual elements or the maximum, minimum, median or mean of all its elements.

⁶ E.g. the sum, difference or product of two data arrays.

less. At first glance this seems promising. But with an increasing number of nodes that participate in one operation the computational efficiency may decrease considerably. This happens if the cost of the actual numerical operations becomes comparable to the generic program and inter-node communication overhead. The ratios highly depend on the specific cluster hardware and the array operations performed.

Problem sizes

Due to our background in signal reconstruction and grid-based simulations, we decide to use low-level parallelization for the following reasons. First, we have to speed up problems that one cannot parallelize algorithmically, like fixed-point iterations or step-wise simulations. Second, we want to scale our algorithms to higher resolutions while keeping the computing time at least constant. Thereby the involved data arrays become so big that a single computer would be oversubscribed. Because of this, the ratio of *array size* to *desired degree of parallelization* does not become such that the computational efficiency would decrease considerably. In practice we experience a good scaling behavior with up to $\approx 10^3$ processes⁷ for problems of size 8192^2 , cf. "Performance and scalability" section. Hence, for our applications the advantages of low-level parallelization clearly outweigh its drawbacks.

D2O as layer of abstraction

Compared to high-level parallelization, the low-level approach is more complicated to implement. In the best case, for the former one simply runs the serial code in parallel on the individual machines; when finished one collects and combines the results. For the latter, when doing the explicit coding one deals with local data portions of the global data array on the individual nodes of the cluster. Hence, one has to keep track of additional information: for example, given a distribution scheme, which portion of the global data is stored on which node of the cluster? Keeping the number of cluster nodes, the size and the dimensionality of the data arrays arbitrary implies a considerable complication for indexing purposes. By this, while implementing an application one has to take care of two non-trivial tasks. On the one hand, one must program the logic of distributing and collecting the data; i.e. the data handling. On the other hand, one must implement the application's actual (abstract) algorithm. Those two tasks are conceptually completely different and therefore a mixture of implementations should be avoided. Otherwise there is the risk that features of an initial implementation—like the data distribution scheme—become hard-wired to the algorithm, inhibiting its further evolution. Thus it makes sense to insert a layer of abstraction between the algorithm code and the data distribution logic. Then the abstract algorithm can be written in a serial style from which all knowledge and methodology regarding the data distribution is encapsulated. This layer of abstraction is D2O.

Choosing a parallelization environment

To make the application spectrum of D2O as wide as possible we want to maximize its portability and reduce its dependencies. This implies that—despite its parallel architecture—D2O must just as well run within a single-process environment for cases when no elaborate parallelization back end is available. But nevertheless, D2O must be massively

⁷ This was the maximum number of processes available for testing.

scalable. This relates to the question of which distribution environment should be used. There are several alternatives:

- Threading and multiprocessing: These two options limit the application to a single machine which conflicts with the aim of massive scalability.
- (py)Spark [16] and hadoop [17]: These modern frameworks are very powerful but regrettably too abstract for our purposes, as they prescind the location of individual portions of the full data. Building a numpy-like interface would be disproportionately hard or even unfeasible. In addition to that, implementing a low-level interface for highly optimized applications which interact with the node's local data is not convenient within pySpark. Lastly, those frameworks are usually not installed as standard dependencies on scientific HPC clusters.
- MPI [5, 6]: The *Message Passing Interface* is available on virtually every HPC cluster via well-tested implementations like *OpenMPI* [18], *MPICH2* [19] or *Intel MPI* [20]. The open implementations are also available on commodity multicore hardware like desktops or laptops. A Python interface to MPI is given by the Python module *mpi4py* [21]. MPI furthermore offers the right level of abstraction for hands-on control of distribution strategies for the package developers.

Given these features we decide to use *MPI* as the parallelization environment for *D2O*. We stress that in order to fully utilize *D2O* on multiple cores, a user does not need to know how to program in *MPI*; it is only necessary to execute the program via *MPI* as shown in the example in "[Distributed arrays](#)" section.

Internal structure

Composed object

A main goal for the design of *D2O* was to make no a-priori assumptions about the specific distribution strategies that will be used in order to spread array data across the nodes of a cluster. Because of this, *D2O*'s distributed array—`d2o.distributed_data_object`—is a composed object; cf. Fig. 1.

The *distributed_data_object* itself provides a rich user interface, and makes sanity and consistency checks regarding the user input. In addition to that, the *distributed_data_object* possesses an attribute called `data`. Here the *MPI* processes' local portion of the global array data is stored, even though the *distributed_data_object* itself will never make any assumptions about its specific content since the distribution strategy is arbitrary in the first place. The *distributed_data_object* is the only object of the *D2O* library that a casual user would interact with.

For all tasks that require knowledge about the certain distribution strategy every *distributed_data_object* possesses an instance of a `d2o.distributor` subclass. This object stores all the distribution-scheme and cluster related information it needs in order to scatter (gather) data to (from) the nodes and to serve for special methods, e.g. the array-cumulative sum. The *distributed_data_object* builds its rich user interface on top of those abstracted methods of its distributor.

The benefit of this strict separation is that the user interface becomes fully detached from the distribution strategy; may it be block-cyclic or slicing, or have neighbor ghost

cells or not, et cetera. Currently there are two fundamental distributors available: a generic *slicing*⁸ and a *not*-distributor. From the former, three special slicing distributors are derived: *fftw*⁹, *equal*¹⁰ and *freeform*.¹¹ The latter, the *not*-distributor, does not do any data-distribution or -collection but stores the full data on every node redundantly.

Advantages of a global view interface

D2O's global view interface makes it possible to build software that remains completely independent from the distribution strategy and the used number of cluster processes. This in turn enables the development of 3rd party libraries that are very end-use-case independent. An example for this may be a mathematical optimizer; an object which tries to find for a given scalar function f an input vector \vec{x} such that the output $y = f(\vec{x})$ becomes minimal. It is interesting to note that many optimization algorithms solely use basic arithmetics like vector addition or scalar multiplication when acting on \vec{x} . As such operations act locally on the elements of an array, there is no preference for one distribution scheme over another when distributing \vec{x} among nodes in a cluster. Two different distribution schemes will yield the same performance if their load-balancing is on a par with each other. Further assume that f is built on D2O, too. On this basis, one could now build an application that uses the minimizer but indeed has a preference for a certain distribution scheme. This may be the case if the load-balancing of the used operations is non-trivial and therefore only a certain distribution scheme guarantees high evaluation speeds. While the application's developer therefore enforces this scheme, the minimizer remains completely unaffected by this as it is agnostic of the array's distribution strategy.

Basic usage

In the subsequent sections we will illustrate the basic usage of D2O in order to explain its functionality and behavior. A more extended discussion is given in "Appendix 1". Our naming conventions are:

- instances of the `numpy.ndarray` class are labeled `a` and `b`,
- instances of `d2o.distributed_data_object` are labeled `obj` and `p`.

In addition to these examples, the interested reader is encouraged to have a look into the `distributed_data_object` method's docstrings for further information; cf. the project's web page <https://gitlab.mpcdf.mpg.de/ift/D2O>.

Initialization

Here we discuss how to initialize a `distributed_data_object` and compare some of its basic functionality to that of a `numpy.ndarray`. First we import the packages.

⁸ The slicing is done along the first array axis.

⁹ The `fftw`-distributor uses routines from the `pyFFTW` [8, 22] package [8] for the data partitioning.

¹⁰ The `equal`-distributor tries to split the data in preferably equal-sized parts.

¹¹ The local data array's first axis is of arbitrary length for the `freeform`-distributor.

```

1 | In [1]: import numpy as np
2 | In [2]: from d2o import distributed_data_object

```

Now we set up some test data using numpy.

```

1 | In [3]: a = np.arange(12).reshape((3, 4))
2 | In [4]: a
3 | Out [4]: array([[ 0,  1,  2,  3],
4 |             [ 4,  5,  6,  7],
5 |             [ 8,  9, 10, 11]])

```

One way to initialize a *distributed_data_object* is to pass an existing numpy array.

```

1 | In [5]: obj = distributed_data_object(a)
2 | In [6]: obj
3 | Out [6]: <distributed_data_object>
4 |         array([[ 0,  1,  2,  3],
5 |                [ 4,  5,  6,  7],
6 |                [ 8,  9, 10, 11]])

```

The output of the `obj` call shows the local portion of the global data available in this process.

Arithmetics

Simple arithmetics and point-wise comparison work as expected from a numpy array.

```

1 | In [7]: (2*obj, obj**3, obj >= 5)
2 | Out [7]: (<distributed_data_object>
3 |          array([[ 0,  2,  4,  6],
4 |                 [ 8, 10, 12, 14],
5 |                 [16, 18, 20, 22]]),
6 |          <distributed_data_object>
7 |          array([[ 0,  1,  8,  27],
8 |                 [ 64, 125, 216, 343],
9 |                 [ 512, 729, 1000, 1331]]),
10 |          <distributed_data_object>
11 |          array([[False, False, False, False],
12 |                 [False, True, True, True],
13 |                 [ True,  True,  True,  True]], dtype=bool))

```

Please note that the *distributed_data_object* tries to avoid inter-process communication whenever possible. Therefore the returned objects of those arithmetic operations are instances of *distributed_data_object*, too. However, the `D2O` user must be careful when combining *distributed_data_objects* with numpy arrays. If one combines two objects with a binary operator in Python (like `+`, `-`, `*`, `\`, `%` or `**`, it will try to call the respective method (`__add__`, `__sub__`, etc.) of the *first* object. If this fails, i.e. if it throws an exception, Python will try to call the *reverse* methods of the *second* object (`__radd__`, `__rsub__`, etc.):

```

1 | In [8]: a + 1; # calls a.__add__(1) -> returns a numpy array
2 | In [9]: 1 + a; # 1.__add__ not existing -> a.__radd__(1)

```

Depending on the conjunction's ordering, the return type may vary when combining numpy arrays with *distributed_data_objects*. If the numpy array is in the first place, numpy will try

to extract the second object's array data using its `__array__` method. This invokes the *distributed_data_object's* `get_full_data` method that communicates the full data to every process. For large arrays this is extremely inefficient and should be avoided by all means. Hence, it is crucial for performance to assure that the *distributed_data_object's* methods will be called by Python. In this case, the locally relevant parts of the array are extracted from the numpy array and then efficiently processed as a whole.

```

1 | In [10]: a + obj # numpy converts obj -> inefficient
2 | Out[10]: array([[ 0,  2,  4,  6], # note: numpy.ndarray
3 |           [ 8, 10, 12, 14],
4 |           [16, 18, 20, 22]])
5 |
6 | In [11]: obj + a # obj processes a -> efficient
7 | Out[11]: <distributed_data_object>
8 |           array([[ 0,  2,  4,  6],
9 |                 [ 8, 10, 12, 14],
10 |                [16, 18, 20, 22]])

```

Array indexing

The *distributed_data_object* supports most of numpy's indexing functionality, so it is possible to work with scalars, tuples, lists, numpy arrays and *distributed_data_objects* as input data. Every process will extract its locally relevant part of the given data-object and then store it; cf. "Array indexing" section.

```

1 | In [12]: obj
2 | Out[12]: <distributed_data_object>
3 |           array([[ 0,  1,  2,  3],
4 |                 [ 4,  5,  6,  7],
5 |                 [ 8,  9, 10, 11]])
6 |
7 | In [13]: obj[1] # extract a row
8 | Out[13]: <distributed_data_object>
9 |           array([4, 5, 6, 7])
10 |
11 | In [14]: obj[1,-2] # extract single entry
12 | Out[14]: 6
13 |
14 | In [15]: obj[:,2, 1::2] # slicing notation
15 | Out[15]: <distributed_data_object>
16 |           array([[ 1,  3],
17 |                 [ 9, 11]])
18 |
19 | # sets data using slicing
20 | In [16]: obj[:,2, 1::2] = [[111, 222], [333, 444]]
21 | In [17]: obj
22 | Out[17]: <distributed_data_object>
23 |           array([[ 0, 111,  2, 222],
24 |                 [ 4,  5,  6,  7],
25 |                 [ 8, 333, 10, 444]])

```

By default it is assumed that all processes use the *same* key-object when accessing data. See "Local keys" section for more details regarding process-individual indexing.

Distribution strategies

In order to specify the distribution strategy explicitly one may use the “distribution_strategy” keyword:

```

1 | In [18]: obj = distributed_data_object(
2 |         a, distribution_strategy='equal')
3 | In [19]: obj.distribution_strategy
4 | Out[19]: 'equal'

```

See "[Copy methods](#)" section for more information on distribution strategies.

Distributed arrays

To use D2O in a distributed manner, one has to create an MPI job. This example shows how four MPI processes hold individual parts of the global data and how distributed read & write access works. The script is started via the command:

```
mpirun -n 4 python get_set_data.py
```

```

1 | # get_set_data.py
2 | from mpi4py import MPI
3 | import numpy as np
4 | from d2o import distributed_data_object
5 | # Get the process' rank number (0,1,2,3) from MPI
6 | rank = MPI.COMM_WORLD.rank
7 |
8 | # Initialize some data
9 | a = np.arange(16).reshape((4,4))
10 | # Initialize the distributed_data_object
11 | obj = distributed_data_object(a)
12 |
13 | # Print the process' local data
14 | print (rank, obj.get_local_data())
15 | # extract data via slicing
16 | print (rank, obj[0:3:2, 1:3].get_local_data())
17 |
18 | b = -np.arange(4).reshape((2,2))
19 | obj[2:4,1:3] = b # Write b into obj
20 |
21 | # Print the process' local data
22 | print (rank, obj.get_local_data())
23 |
24 | # Consolidate the data
25 | full_data = obj.get_full_data()
26 | if rank == 0: print (rank, full_data)

```

The *distributed_data_object* gets initialized in line 11 with the following array:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Here, the script is run in four MPI processes; cf. `mpirun -n4 [...]`. The data is split along the first axis; the print statement in line 14 yields the four pieces:

```
(0, array([[ 0,  1,  2,  3]]))
(1, array([[ 4,  5,  6,  7]]))
(2, array([[ 8,  9, 10, 11]]))
(3, array([[12, 13, 14, 15]]))
```

The second print statement (line 16) illustrates the behavior of data extraction; `obj[0:3:2, 1:3]` is slicing notation for the entries 1, 2, 9 and 10.¹² This expression returns a *distributed_data_object* where the processes possess the individual portion of the requested data. This means that the distribution strategy of the new (sub-)array is determined by and aligned to that of the original array.

```
(0, array([[1, 2]]))
(1, array([], shape=(0, 2), dtype=int64)) # empty
(2, array([[ 9, 10]]))
(3, array([], shape=(0, 2), dtype=int64)) # empty
```

The result is a *distributed_data_object* where the processes 1 and 3 do not possess any data as they had no data to contribute to the slice in `obj[0:3:2, 1:3]`. In line 19 we store a small 2×2 block `b` in the lower middle of `obj`. The process' local data reads:

```
(0, array([[ 0,  1,  2,  3]]))
(1, array([[ 4,  5,  6,  7]]))
(2, array([[ 8,  0, -1, 11]]))
(3, array([[12, -2, -3, 15]]))
```

Finally, in line 25 we use `obj.get_full_data()` in order to consolidate the distributed data; i.e. to communicate the individual pieces between the processes and merge them into a single numpy array.

```
(0, array([[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  0, -1, 11],
           [12, -2, -3, 15]]))
```

Performance and scalability

In this section we examine the scaling behavior of a *distributed_data_object* that uses the *equal* distribution strategy. The timing measurements were performed on the C2PAP Computing Cluster [23].¹³ The software stack was built upon *Intel MPI 5.1*, *mpi4py 2.0*, *numpy 1.11* and *python 2.7.11*. For measuring the individual timings we used the Python standard library module *timeit* with a fixed number of 100 repetitions.

Please note that `D2O` comes with an extensive suite of unit tests featuring a high code-coverage rate. By this we assure `D2O`'s continuous code development to be very robust and provide the users with a reliable interface definition.

¹² This notation can be decoded as follows. The numbers in a slice correspond to `start:stop:step` with `stop` being exclusive. `obj[0:3:2, 1:3]` means to take every second line from the lines 0, 1 and 2, and to then take from this the elements in columns 1 and 2.

¹³ The C2PAP computing cluster consists of 128 nodes, each possessing two Intel Xeon CPU E5-2680 (8 cores 2.70 GHz + hyper-threading, 64 KiB L1 per core, 256 KiB L2 cache per core, 20 MiB L3 cache shared for all 8 cores) and 64 GiB RAM each. The nodes are connected via Mellanox Infiniband 40 Gbits/s.

Regarding `D2O`'s performance there are two important scaling directions: the size of the array data and the number of MPI processes. One may distinguish three different contributions to the overall computational cost. First, there is data management effort that every process has to cover itself. Second, there are the costs for inter-MPI-process communication. And third, there are the actual numerical costs.

`D2O` has size-independent management overhead compared to `numpy`. Hence, the larger the arrays are for which `D2O` is used, the more efficient the computations become. We will see below that there is a certain array size per node—roughly 2^{16} elements—from which on `D2O`'s management overhead becomes negligible compared to the purely numerical costs. This size corresponds to a two-dimensional grid with a resolution of 256×256 or equivalently 0.5 MiB of 64-bit doubles. In "[Scaling the array size](#)" section we focus on this very ratio of management overhead to numerical costs.

`D2O` raises the claim to be able to operate well running with a single process as well as in a highly parallelized regime. In "[Weak scaling: proportional number of processes and size of data](#)" section, the scaling analysis regarding the MPI process count is done with a fixed local array size for which the process overhead is negligible compared to the numerical costs. For this weak scaling analysis we are interested in the costs arising from inter-process communication compared to those of actual numerics.

In the following three sections, we study the strong scaling of `D2O` where the performance is a result of the combination of all three cost contributions. "[Strong scaling: varying number of processes with a fixed size of data](#)" section covers the case in which the number of MPI processes is increased while the array size is left constant. In "[Strong scaling: comparison with DistArray](#)" section we compare `D2O`'s to `DistArray`'s [7] performance and finally, in section [Strong scaling: real-world application speedup—the Wiener filter](#) we benchmark `D2O`'s strong-scaling behavior when applied to a real-world application: a Wiener filter signal reconstruction.

A discussion on `D2O`'s efficient Python iterators can be found in "[Appendix 2](#)".

Scaling the array size

One may think of `D2O` as a layer of abstraction that is added to `numpy` arrays in order to take care of data distribution and collection among multiple MPI processes. This abstraction comes with inherent Python overhead, separately for each MPI process. Therefore, if one wants to analyze how the ratio of *management overhead* to *actual numerical effort* varies with the data size, only the individual process' data size is important. Because of this, all timing tests for this subsection were carried out with one MPI process only.

A common task during almost all numerical operations is to create a new array object for storing its results.¹⁴ Hence, the speed of object creation can be crucial for overall performance there. Note that in contrast to a `numpy` array which basically just allocates RAM, several things must be done during the initialization of a *distributed_data_object*. The Python object instance itself must be created, a distributor must be initialized which involves parsing of user input, RAM must be allocated, the *distributed_data_object* must be registered with the `d2o_librarian` (cf. "[The d2o librarian](#)" section), and, if activated,

¹⁴ Exceptions to this are inplace operations which reuse the input array for the output data.

inter-MPI-process communication must be done for performing sanity checks on the user input.

By default the initialization requires 60 μ s to complete for a *distributed_data_object* with a shape of (1,) when run within one single MPI process. Using this trivial shape makes the costs for memory allocation negligible compared to the others tasks. Hence, those 60 μ s represent D2O's constant overhead compared to numpy, since a comparable numpy array requires ≈ 0.4 μ s for initialization.

In order to speed up the initialization process one may disable all sanity checks on the user input that require MPI communication, e.g. if the same datatype was specified in all MPI processes. Even when run with one single process, skipping those checks reduces the costs by 27 μ s from 60 to 33 μ s.

Because of the high costs, it is important to avoid building *distributed_data_objects* from scratch over and over again. A simple measure against this is to use inplace operations like `obj += 1` instead of `obj = obj + 1` whenever possible. This is generally a favorable thing to do—also for numpy arrays—as this saves the costs for repeated memory allocation. Nonetheless, also non-inplace operations can be improved in many cases, as often the produced and the initial *distributed_data_object* have all of their attributes in common, except for their data: they are of the same shape and datatype, and use the same distribution strategy and MPI communicator; cf. `p = obj + 1`. With `obj.copy()` and `obj.copy_empty()` there exist two cloning methods that we implemented to be as fast as allowed by pure Python. Those methods reuse as much already initialized components as possible and are therefore faster than a fresh initialization: for the *distributed_data_object* from above `obj.copy()` and `obj.copy_empty()` consume 7.9 and 4.3 μ s, respectively.

Table 3 shows the performance ratio in percent between serial D2O and numpy. The array sizes range from $2^0 = 1$ to $2^{25} \approx 3.3 \cdot 10^7$ elements. In the table, 100 % would mean that D2O is as fast as numpy.

The previous section already suggested that for tasks that primarily consist of initialization work—like *array creation* or `copy_empty`—D2O will clearly follow behind numpy. However, increasing the array size from 2^{20} to 2^{22} elements implies a considerable performance drop for numpy's memory allocation. This in turn means that for arrays with more than 2^{22} elements D2O's relative overhead becomes less significant: e.g. `np.copy_empty` is then only a factor of four faster than `obj.copy_empty()`.

Functions like `max` and `sum` return a scalar number; no expensive return-array must be created. Hence, D2O's overhead is quite modest: even for size 1 arrays, D2O's relative performance lies above 50 %. Once the size is greater than 2^{18} elements the performance is higher than 95 %.

`obj[::-2]` is *slicing syntax* for “take every second element from the array in reverse order”. It illustrates the costs of the data-distribution and collection logic that even plays a significant role if there is no inter-process communication involved. Again, with a large-enough array size, D2O's efficiency becomes comparable to that of numpy.

Similarly to `obj[::-2]`, the remaining functions in the table return a *distributed_data_object* as their result and therefore suffer from its initialization costs. However, with an array size of 2^{16} elements and larger D2O's relative performance is at least greater than approximately 65 %.

An interesting phenomenon can be observed for `obj + 0` and `obj + obj`: As for the other functions, their relative performance starts to increase significantly when an array size of 2^{16} is reached. However, in contrast to `obj += obj` which then immediately scales up above 95 %, the relative performance of the non-inplace additions temporarily *decreases* with increasing array size. This may be due to the fact that given our test scenario 2^{18} elements almost take up half of the cache of C2PAP's Intel E5-2680 CPUs. D2O's memory overhead is now responsible for the fact, that its non-inplace operations—which need twice the initial memory—cannot profit that strongly from the cache anymore, whereas the numpy array still operates fast. Once the array size is above 2^{22} elements numpy's just as D2O's array-object is too large for profiting from the cache and therefore become comparably fast again: the relative performance is then greater than 98 %.

Thus, when run within a single process, D2O is ideally used for arrays larger than $2^{16} = 65536$ elements which corresponds to 512 KiB. From there the management overhead becomes less significant than the actual numerical costs.

Weak scaling: proportional number of processes and size of data

Now we analyze the scaling behavior of D2O when run with several MPI processes. Repeating the beginning of "[Performance and scalability](#)" section, there are three contributions to the execution time. First, the fixed management overhead that every process has to cover itself, second, the communication overhead and third, the actual numerical costs. In order to filter out the effect of a changing contribution of management overhead, in this section we fix the MPI processes' local array size to a fixed value. Hence, now the global data size is proportional to the number of MPI processes.

Table 4 shows the performance of various array operations normalized to the time D2O needs when running with one process only. Assuming that D2O had no communication overhead and an operation scaled perfectly linearly with array size, the performance would be rated at 100 %.

In theory, operations that do not inherently require inter-process communication like point-wise array addition or subtraction ideally scale linearly. And in fact, D2O scales excellently with the number of processes involved for those functions: here we tested `copy`, `copy_empty`, `sum(axis = 1)`, `obj + 0`, `obj + obj`, `obj += obj` and `sqrt`.

Comparing `sum(axis = 0)` with `sum(axis = 1)` illustrates the performance difference between those operations that involve inter-process communication and those that don't: the *equal* distribution strategy slices the global array along its first axis in order to distribute the data among the individual MPI processes. Hence, `sum(axis = 0)`—which means to take the sum along the first axis—does intrinsically involve inter-process communication whereas `sum(axis = 1)` does not. Similarly to `sum(axis = 0)` also the remaining functions in Table 4 are affected by an increasing number of processes as they involve inter-process communication.

But still, even if—for example in case of `sum(axis = 0)`—the relative performance may drop to 28.2 % when using 256 processes, this means that the operation just took

3.5 times longer than the single-process run, whereat the array size has been increased by a factor of 256. This corresponds to a speedup factor of 72.2.

Strong scaling: varying number of processes with a fixed size of data

Similarly to the previous "Weak scaling: proportional number of processes and size of data" section, we vary the number of processes but now fix the data size *globally* instead of locally. This corresponds to the real-life scenario in which the problem size and resolution are already set—maybe by environmental conditions—and now one tries to reduce the run time by using more cores. Since the size of the local data varies with the number of processes, the overall scaling behavior is now a mixture of the varying ratio between management overhead and process-individual numerical costs, and the fact that an increasing amount of CPU cache becomes available at the expense of increased communication effort. Table 5 shows the benchmarking results for the same set of operations as used in the previous section on weak scaling and the results are reasonable. Those operations in the list that inherently cannot scale strongly as they consist of purely node-individual work, namely the `initialization` and `copy_empty`, show that their performance just does not increase with the number of processes. In contrast, operations without communication effort benefit strongly from the increasing total amount of CPU cache combined with smaller local arrays; above all `copy` which is about 3 times faster than what one would expect from linear scaling to 256 processes.¹⁵

In theory, the strong-scaling behavior is the combination of the size- and weak-scaling we discussed in sections "Scaling the array size" and "Weak scaling: proportional number of processes and size of data". In order to verify whether the strong-scaling behavior makes sense, we estimate the strong-scaling performance using the information from size- and weak-scaling.

We choose the `sum()` method as our test case. During the reduction phase, the n MPI-processes exchange their local results with each other. This corresponds to adding n times a fixed amount of communication time to the total computing time needed. Hence, we perform a linear fit to the weak-scaling data; cf. Table 4. Furthermore, we assume that the local computation time is roughly proportional to the local-array size. This is true for sufficiently large array sizes, since then `numpy` scales linearly and `D2O` is in a good efficiency regime, cf. Table 3. Again we performed a linear fit but now on the size-scaling timing data. Combining those two linear fits leads to the following run-time formula for applying `sum()` to an array with shape (4096, 4096):

$$t(n) = (0.0065n + 1.57/n) \text{ s} \quad (1)$$

In the case of linear scaling, $t(n)$ is expected to be equal to $t(1)/n$. Hence, the relative performance $p(n)$ is the ratio between the two:

$$p(n)_{\text{estimated}} = \frac{\frac{t(1)}{n}}{t(n)} = \frac{241.8}{240.8 + n^2} \quad (2)$$

¹⁵ This very strong scaling is indeed realistic: when analyzing pure `numpy` arrays one gets speedups in the order of magnitude of even 800 %.

Table 1 Strong scaling behavior: estimate vs. measurement

#processes : n	1 (%)	4 (%)	8 (%)	16 (%)	32 (%)	64 (%)	128 (%)	256 (%)
$\rho(n)_{\text{estimated}}$	100	94.2	79.3	48.7	19.1	5.58	1.45	0.37
$\rho(n)_{\text{measured}}$	100	91.7	74.1	60.9	24.4	7.05	1.82	0.45

Table 2 Execution time scaling of a Wiener filter reconstruction on a grid of size 8192 × 8192

#nodes	1	1	1	1	2	4	8	16	32
#processes : n	1	2	3	4	8	16	32	64	128
t [S]	1618	622.0	404.2	364.2	181.7	94.50	46.79	18.74	8.56
$s_n = 2t_2/t_n$	0.769	2.00	3.08	3.42	6.85	13.2	26.6	66.4	145
$q_n = 1/(1 + \log(\frac{n}{s_n}))$	0.900	1.00	1.01	0.94	0.94	0.92	0.93	1.02	1.06

Comparing the estimate with the actually measured relative performance—cf. Tables 1, 2, 3, 4, 5 shows that even under those rough assumptions the strong scaling behavior of `sum()` can be explained as the combination of size- and weak scaling within about 20 % accuracy.

Strong scaling: comparison with `distArray`

In "Alternative packages" section we discussed several competitors to `D2O`. Because of their similarities, we conducted the strong scaling tests—as far as possible¹⁶—also with `DistArray` and compare the performance. In Table 6 the results are shown for the subset of all operations from the previous sections that were available for `DistArray`, too.

While being at least on a par for numerical operations when being run single-threaded, `D2O` outperforms `DistArray` more and more with an increasing number of processes. Furthermore, it is conspicuous that `DistArray` does not seem to support inplace array operations. Because of this, the inplace addition `obj += obj` is way slower with `DistArray` than with `D2O` which is on a par with `numpy` in most cases, cf. Tables 3, 4 and 5.

The fact that `D2O` is way more efficient when doing numerics on very small arrays—like `obj + 0` using 256 processes—indicates that `D2O`'s organizational overhead is much smaller than that of `DistArray`. Supporting evidence for this is that the initialization of an empty `DistArray` (`copy_empty`) becomes disproportionately costly when increasing the number of processes used.

Strong scaling: real-world application speedup—the Wiener filter

`D2O` was initially developed for `NIFTY` [3], a library for building signal inference algorithms in the framework of *information field theory* (IFT) [24]. Within `NIFTY` v2 all of the parallelization is done via `D2O`; the code of `NIFTY` itself is almost completely agnostic of the parallelization and completely agnostic of `MPI`.

¹⁶ `DistArray` does, for example, not support negative step-sizes for slicing (`[:-2]`) and also the special method `bincount` is not available.

Table 3 Overhead costs: d2o's relative performance to numpy when scaling the array size and being run with one MPI-process

Array size	2 ⁰ (%)	2 ² (%)	2 ⁴ (%)	2 ⁶ (%)	2 ⁸ (%)	2 ¹⁰ (%)	2 ¹² (%)	2 ¹⁴ (128 KiB) (%)	2 ¹⁶ (%)	2 ¹⁸ (%)	2 ²⁰ (8 MiB) (%)	2 ²² (%)	2 ²³ (%)	2 ²⁴ (128 MiB) (%)	2 ²⁵ (%)
initialization	0.65	0.64	0.69	0.69	0.71	0.71	0.74	0.72	0.75	0.74	0.75	5.41	5.58	5.83	6.00
copy_empty	3.77	3.86	4.00	4.12	4.57	3.92	3.95	3.98	4.01	4.10	4.15	25.0	24.2	25.3	26.0
max	56.2	56.0	54.4	55.5	56.0	56.9	62.6	79.2	91.7	97.5	99.4	99.4	99.9	99.8	99.9
sum	59.7	59.0	57.5	60.1	58.6	59.8	62.5	74.8	88.4	95.9	99.0	99.3	99.7	99.7	100.0
obj[::2]	1.18	1.17	1.20	1.24	1.34	1.50	2.14	4.77	15.0	22.2	28.7	24.1	45.8	47.3	47.7
copy	8.16	8.86	9.30	9.72	9.66	10.4	14.8	26.7	65.8	95.5	98.7	99.1	99.9	99.4	98.4
obj+0	6.58	6.48	6.67	7.08	7.37	9.51	16.2	35.0	65.2	34.7	44.7	98.7	99.7	96.5	100.0
obj+obj	3.07	3.10	3.24	3.59	3.98	5.70	13.3	31.9	64.0	34.6	45.0	98.7	99.9	99.5	99.8
obj += obj	5.17	5.35	5.41	6.02	6.66	11.3	22.8	46.4	75.3	91.9	97.8	98.0	99.8	99.3	99.7
sqrt	3.25	3.17	3.26c	3.50	4.42	7.49	18.7	45.6	75.8	65.2	83.2	98.8	99.4	99.6	99.8
bincount	3.57	3.35	3.76	4.04	4.97	7.54	16.5	35.4	58.0	75.1	78.0	76.8	78.9	82.1	83.2

"100%" corresponds to the case when d2o is as fast as numpy. In order to guide the eye, values <30% are printed italic, values ≥90% are printed bold-italics. Please see "Scaling the array size" section for discussion

Table 4 Weak scaling: d2o's relative performance to the single-process case when increasing both, the number of processes and the global array size proportionally

Process count	1 (%)	2 (%)	3 (%)	4 (%)	5 (%)	16 (%)	32 (%)	64 (%)	128 (%)	256 (%)
initialization	100.0	90.9	87.9	87.8	74.6	67.6	54.9	45.7	34.6	19.9
copy.empty	100.0	97.5	96.2	97.5	97.6	103.6	97.8	97.7	97.6	95.1
max	100.0	97.5	96.6	95.6	90.9	84.0	72.1	56.2	39.1	24.3
sum	100.0	98.0	95.3	93.5	87.3	79.2	65.1	48.3	32.2	19.2
sum(axis=0)	100.0	100.2	96.7	96.5	90.9	78.1	74.6	58.0	42.7	28.2
sum(axis=1)	100.0	105.2	103.2	102.2	100.6	100.0	98.3	95.8	93.2	88.6
obj[:-2]	100.0	70.4	65.9	64.0	46.2	46.6	42.8	33.6	31.1	25.3
copy	100.0	104.7	103.1	101.3	101.3	105.3	101.4	101.2	101.3	101.5
obj + 0	100.0	105.1	102.6	100.6	99.9	103.5	100.2	100.0	99.7	100.1
obj + obj	100.0	105.2	102.5	100.1	100.0	103.7	100.1	100.1	99.8	100.2
obj + = obj	100.0	102.3	99.3	98.6	98.2	101.8	98.2	98.2	98.2	98.4
sqrt	100.0	102.0	100.6	100.1	99.6	99.1	99.2	99.2	8.6	98.0
bincount	100.0	103.0	101.2	99.9	98.8	97.6	94.1	88.3	79.4	65.8

The arrays used for this tests had the global shape ($n * 2048, 2048$) with n being the number of processes. By this the local data size was fixed to 2^{22} elements, which is equal to 32 MiB. "100 %" in the table corresponds to the case were the speedup is equal to the number of processes. Example: the 95.1 % for `copy.empty` on 256 processes correspond to a speedup-factor of 243.5. In order to guide the eye, values < 30 % are printed italic, values \geq 90 % are printed bold-italics. Please see "[Weak scaling: proportional number of processes and size of data](#)" section for discussion

A standard computational operation in IFT-based signal reconstruction is the *Wiener filter* [25]. For this performance test, we use a Wiener filter implemented in NIFTy to reconstruct the realization of a Gaussian random field—called the *signal*. Assume we performed a hypothetical measurement which produced some *data*. The data model is

$$\text{data} = R(\text{signal}) + \text{noise} \quad (3)$$

where R is a smoothing operator and *noise* is additive Gaussian noise. In Fig. 2 one sees the three steps:

- the true signal we try to reconstruct,
- the data one gets from the hypothetical measurement, and
- the reconstructed signal field that according to the Wiener filter has most likely produced the *data*.

Table 2 shows the scaling behavior of the reconstruction code, run with a resolution of 8192×8192 . Here, n is the number of used processes and t_n the respective execution time. The relative speedup $s_n = 2t_2/t_n$ ¹⁷ is the ratio of execution times: parallel versus serial. In the case of exactly linear scaling s_n is equal to n . Furthermore we define the scaling quality $q = 1/(1 + \log(n/s_n))$, which compares s_n with linear scaling in terms of orders of magnitude. A value $q = 1$ represents linear scaling and $q \geq 1$ super-linear scaling.

This benchmark illustrates that even in a real-life application super-linear scaling is possible to achieve for a sufficiently large number of processes. This is due to the operations that are needed in order to perform the Wiener filtering: basic point-wise

¹⁷ Since the combination of NIFTy and pyfftw exhibits an unexpected speed malus for one process, we chose the two-process timing as the benchmark's baseline.

Table 5 Strong scaling: d2o's relative performance to a single process when increasing the number of processes while fixing the global array size to (4096, 4096) = 128 MiB

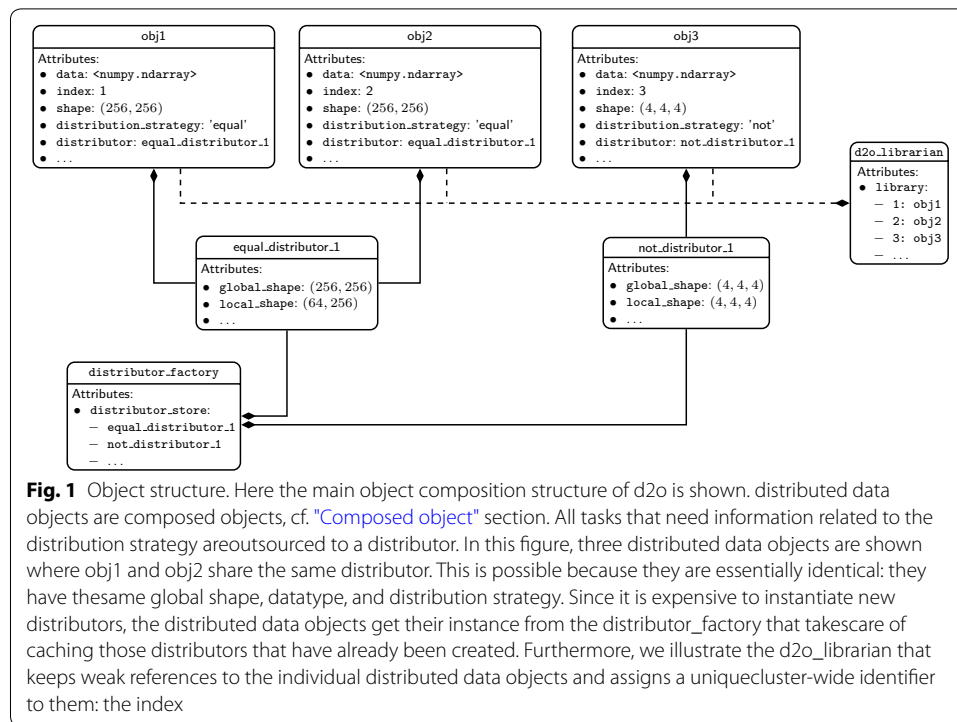
Processes (local size)	1 (128 MiB) (%)	2 (64 MiB) (%)	3 (42.7 MiB) (%)	4 (32 MiB) (%)	8 (16 MiB) (%)	16 (8 MiB) (%)	32 (4 MiB) (%)	64 (2 MiB) (%)	128 (1 MiB) (%)	256 (512 KiB) (%)
initialization	100.0	40.3	23.7	17.10	6.80	2.55	0.89	0.29	0.10,	0.03
copy .empty	100.0	47.8	33.5	26.3	23.2	12.1	6.16	3.09	1.55	0.79
max	100.0	99.0	96.7	94.0	80.1	64.3	29.6	9.10	2.41	0.60
sum	100.0	100.4	95.4	91.7	74.1	60.9	24+4	7.05	1.82	0.45
sum(axes=0)	100.0	98.0	94.2	89.9	62.2	45.3	19.8	6.34	1.78:	0.45
sum(axes=1)	100.0	100.5	92.3	92.6	79.0	77.7	47.1	20.6	4.27	1.25
obj[;-2]	100.0	65.4	62.4	58.5	40.7	33.1	26.6	18.3	8.78	3.25
copy	100.0	103.6	105.9	98.0	145.1	156.4	155.3	152.3	157.5	306.7
obj + 0	100.0	105.9	109.1	100.4	59.0	97.3	75.7	79.2	79.4	149.8
obj + obj	100.0	106.2	109.2	100.3	59.0	97.6	74.9	79.0	80.1	150.2
obj + = obj	100.0	103.1	101.3	98.0	97.8	124.0	122.8	117.9	108.1	94.4
sqrt	100.0	101.8	99.4	98.7	95.7	88.8	76.0	56.0	37.1	16.4
bincount	100.0	102.3	99.5	98.0	111.1	107.3	84.2	40.5	13.2	3.57

"100%" corresponds to the case where the speedup is equal to the number of processes. Example: the 94.4% for obj += obj on 256 processes correspond to a speedup-factor of 241.7. In order to guide the eye, values <30% are printed italic, values ≥90% are printed bold-italics. Please see "Strong scaling: varying number of processes with a fixed size of data" section for discussion

Table 6 Strong scaling comparison: d2o's relative performance compared to DistArray[7] when increasing the number of processes while fixing the global array size to (4096, 4096) = 128 MiB

Processes (local size)	1 (128 MiB)	2 (64 MiB)	3 (42.7 MiB)	4 (32 MiB)	8 (16 MiB)	16 (8 MiB)	32 (4 MiB)	64 (2 MiB)	128 (1 MiB)	256 (512 KiB)
copy_empty	23.49	27.70	33.99	40.03	1.11 × 10 ²	1.12 × 10 ³	1.06 × 10 ³	1.91 × 10 ³	5.57 × 10 ³	1.90 × 10 ⁴
Max	1.05	1.11	1.16	1.20	1.34	5.00	4.74	3.77	3.25	4.17
Sum	1.07	1.20	1.25	1.33	1.48	6.57	5.57	4.20	3.55	4.61
sum(axis=0)	1.02	1.09	1.12	1.22	1.03	3.61	3.35	2.75	2.42	3.06
sum(axis=1)	1.03	1.15	1.15	1.28	1.63	11.35	12.56	19.29	22.55	42.48
obj + 0	1.02	1.09	1.20	1.15	0.44	2.88	4.24	7.81	33.94	3.78 × 10 ²
obj + obj	1.02	1.09	1.20	1.16	0.44	2.90	4.23	8.08	34.40	3.81 × 10 ²
obj + = obj	2.27	2.38	2.51	2.53	1.60	8.23	14.89	26.24	1.04 × 10 ²	5.35 × 10 ²
Sqrt	1.00	1.03	1.03	1.04	0.81	1.57	2.06	2.59	6.66	16.77

"2" corresponds to the case where d2o is twice as fast as DistArray. Please see "Strong scaling: comparison with DistArray" section for discussion



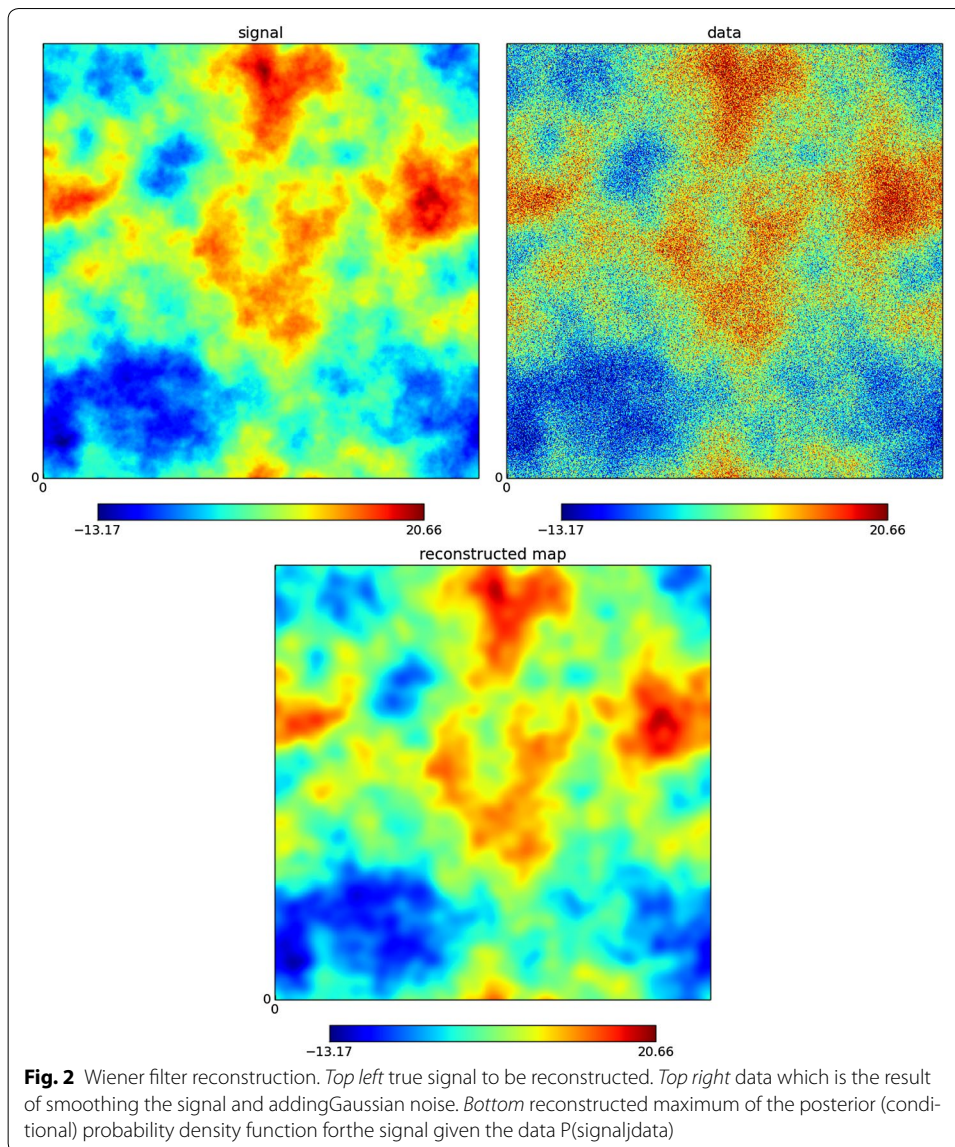
arithmetics that do not involve any inter-process communication and Fourier transformations that are handled by the high-performance library *FFTW* [26]. While the problem size remains constant, the amount of available CPU cache increases with the number of processes, which explains the super-linear scaling, cf. "Strong scaling: varying number of processes with a fixed size of data" section.

Summary and outlook

We introduced *D2O*, a Python module for cluster-distributed multi-dimensional numerical arrays. It can be understood as a layer of abstraction between abstract algorithm code and actual data-distribution logic. We argued why we developed *D2O* as a package following a low-level parallelization ansatz and why we built it on MPI. Compared to other packages available for data parallelization, *D2O* has the advantage of being ready for action on one as well as several hundreds of CPUs, of being highly portable and customizable as it is built with Python, that it is faster in many circumstances, and that it is able to treat arrays of arbitrary dimension.

For the future, we plan to cover more of *numpy*'s interface such that working with *D2O* becomes even more convenient. Furthermore we evaluate the option to build a *D2O* distributor in order to support *scalapy*'s block-cyclic distribution strategy directly. This will open up a whole new class of applications *D2O* then can be used for.

D2O is open source software licensed under the GNU General Public License v3 (GPL-3) and is available by <https://gitlab.mpcdf.mpg.de/ift/D2O>.



Authors' contributions

TS is the principal researcher for the work proposed in this article. His contributions include the primal idea, the implementation of presented software package, the conduction of the performance tests and the writing of the article. MG, FB and TE helped working out the conceptual structure of the software package and drafting the manuscript. FB also played a pivotal role for executing the performance tests. Furthermore, TE also fulfilled the role of the principal investigator. All authors read and approved the final manuscript.

Author details

¹ Max Planck Institut für Astrophysik, Karl-Schwarzschild-Strasse 1, 85741 Garching, Germany. ² Ludwig-Maximilians-Universität München, Geschwister-Scholl-Platz 1, 80539 München, Germany. ³ Exzellenzcluster Universe, Boltzmannstrasse 2, 85748 Garching, Germany.

Acknowledgements

We want to thank Jait Dixit, Philipp Franck, Reimar Leike, Fotis Megas, Martin Reinecke and Csongor Varady for useful discussions and support. We acknowledge the support by the DFG Cluster of Excellence “Origin and Structure of the Universe” and the Studienstiftung des deutschen Volkes. The performance tests have been carried out on the computing facilities of the Computational Center for Particle and Astrophysics (C2PAP). We are grateful for the support by Dr. Alexey Krukau through the Computational Center for Particle and Astrophysics (C2PAP).

Competing interests

The authors declare that they have no competing interests.

Appendix 1: Advanced usage and functional behavior

Here we discuss specifics regarding the design and functional behavior of `D2O`. We set things up by importing `numpy` and `d2o.distributed_data_object`:

```
1 | In [1]: import numpy as np
2 | In [2]: from d2o import distributed_data_object
```

Distribution strategies

In order to see the effect of different distribution strategies one may run the following script using three MPI processes. In lines 13 and 19, the `distribution_strategy` keyword is used for explicit specification of the strategy.

```
mpirun -n 3 python distribution_schemes.py
```

```
1 | # distribution_schemes.py
2 | from mpi4py import MPI
3 | import numpy as np
4 | from d2o import distributed_data_object
5 | rank = MPI.COMM_WORLD.rank
6 |
7 | a = np.arange(16).reshape((4, 4))
8 | if rank == 0: print((rank, a))
9 |
10 | # use 'not', 'equal' and 'fftw'
11 | for strategy in ['not', 'equal', 'fftw']:
12 |     obj = distributed_data_object(
13 |         a, distribution_strategy=strategy)
14 |     print (rank, strategy, obj.get_local_data())
15 |
16 | # use the 'freeform' slicer
17 | a += rank
18 | obj = distributed_data_object(
19 |     local_data=a, distribution_strategy='freeform')
20 | print (rank, 'freeform', obj.get_local_data())
21 |
22 | full_data = obj.get_full_data()
23 | if rank == 0: print (rank, 'freeform', full_data)
```

The printout in line 8 shows the `a` array.

```
(0, array([[ 0,  1,  2,  3],
          [ 4,  5,  6,  7],
          [ 8,  9, 10, 11],
          [12, 13, 14, 15]]))
```

The “not” distribution strategy stores full copies of the data on every node:

```
(0, 'not', array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11],
                 [12, 13, 14, 15]]))
(1, 'not', array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11],
                 [12, 13, 14, 15]]))
(2, 'not', array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11],
                 [12, 13, 14, 15]]))
```

The “equal”, “fftw” and “freeform” distribution strategies are all subtypes of the *slicing* distributor that cuts the global array along its first axis. Therefore they only differ by the lengths of their subdivisions. The “equal” scheme tries to distribute the global array as equally as possible among the processes. If the array’s size makes it necessary, the first processes will get an additional row. In this example the first array axis has a length of four but there are three MPI processes; hence, one gets a distribution of (2, 1, 1):

```
(0, 'equal', array([[0, 1, 2, 3],
                  [4, 5, 6, 7]]))
(1, 'equal', array([[ 8,  9, 10, 11]]))
(2, 'equal', array([[12, 13, 14, 15]]))
```

The “fftw” distribution strategy is very similar to “equal” but uses functions from FFTW[8]. If the length of the first array axis is large compared to the number of processes they will practically yield the same distribution pattern but for small arrays they may differ. For performance reasons FFTW prefers multiples of two over a uniform distribution, hence one gets (2, 2, 0):

```
(0, 'fftw', array([[0, 1, 2, 3],
                  [4, 5, 6, 7]]))
(1, 'fftw', array([[ 8,  9, 10, 11],
                  [12, 13, 14, 15]]))
(2, 'fftw', array([], shape=(0, 4), dtype=int64))
```

A “freeform” array is built from a process-local perspective: each process gets its individual local data. In our example, we use `a+rank` as the local data arrays—each being of shape (4, 4)—during the initialization of the *distributed_data_object*. By this, a global shape of (12, 4) is produced. The local data reads:


```
(0, 'freeform', array([[ 0,  1,  2,  3],
                       [ 4,  5,  6,  7],
                       [ 8,  9, 10, 11],
                       [12, 13, 14, 15]]))
(1, 'freeform', array([[ 1,  2,  3,  4],
                       [ 5,  6,  7,  8],
                       [ 9, 10, 11, 12],
                       [13, 14, 15, 16]]))
(2, 'freeform', array([[ 2,  3,  4,  5],
                       [ 6,  7,  8,  9],
                       [10, 11, 12, 13],
                       [14, 15, 16, 17]]))
```

This yields a global shape of (12, 4). In order to consolidate the data the method `obj.get_full_data()` is used, cf. "[Getting and setting data](#)" section.

```
(0, 'freeform', array([[ 0,  1,  2,  3],
                       [ 4,  5,  6,  7],
                       [ 8,  9, 10, 11],
                       [12, 13, 14, 15],
                       [ 1,  2,  3,  4],
                       [ 5,  6,  7,  8],
                       [ 9, 10, 11, 12],
                       [13, 14, 15, 16],
                       [ 2,  3,  4,  5],
                       [ 6,  7,  8,  9],
                       [10, 11, 12, 13],
                       [14, 15, 16, 17]]))
```

Initialization

There are several different ways of initializing a *distributed_data_object*. In all cases its shape and data type must be specified implicitly or explicitly. In the previous section we encountered the basic way of supplying an initial data array which then gets distributed:

```
1 | In [3]: a = np.arange(12).reshape((3, 4))
2 | In [4]: obj = distributed_data_object(a)
3 | # equivalent to line above
4 | In [5]: obj = distributed_data_object(global_data=a)
```

The initial data is interpreted as global data. The default distribution strategy¹⁸ is a *global-type* strategy, which means that the distributor which is constructed at initialization time derives its concrete data partitioning from the desired global shape and data type. A more explicit example for an initialization is:

¹⁸ Depending on whether *pyfftw* is available or not, the *equal-* or the *fftw-*distribution strategy is used, respectively; cf. "[Distribution strategies](#)" section.

```

1 | In [6]: obj = distributed_data_object(global_data=a,
2 |                                     dtype=np.complex)

```

In contrast to `a`'s data type which is *integer* we enforce the *distributed_data_object* to be *complex*. Without initial data—cf. `np.empty`—one may use the `global_shape` keyword argument:

```

1 | In [7]: obj = distributed_data_object(global_shape=(2,3),
2 |                                     dtype=np.float)
3 | # equivalent to line above
4 | In [8]: obj = distributed_data_object(global_shape=(2,3))

```

If the data type is specified neither implicitly by some initial data nor explicitly via `dtype`, *distributed_data_object* uses `float` as a default¹⁹. In contrast to *global-type*, *local-type* distribution strategies like “freeform” are defined by local shape information. The aptly named analoga to `global_data` and `global_shape` are `local_data` and `local_shape`, cf. “[Distribution strategies](#)” section :

```

1 | In [9]: obj = distributed_data_object(
2 |         local_data=a,
3 |         distribution_strategy='freeform')

```

If redundant but conflicting information is provided—like integer-type initialization array vs. `dtype=complex`—the explicit information gained from `dtype` is preferred over implicit information provided by `global_data/local_data`. On the contrary, if data is provided, explicit information from `global_shape/local_shape` is discarded. In summary, `dtype` takes precedence over `global_data/local_data` which in turn takes precedence over `global_shape/local_shape`.

Please note that besides numpy arrays, *distributed_data_objects* are valid input for `global_data/local_data`, too. If necessary, a redistribution of data will be performed internally. When using `global_data` this will be the case if the distribution strategies of the input and output *distributed_data_objects* do not match. When *distributed_data_objects* are used as `local_data` their full content will be concentrated on the individual processes. This means that if one uses the same *distributed_data_object* as `local_data` in, for example, two processes, the resulting *distributed_data_object* will have twice the memory footprint.

Getting and setting data

There exist three different methods for getting and setting a *distributed_data_object*'s data:

- `get_full_data` consolidates the full data into a numpy array,
- `set_full_data` distributes a given full-size array,

¹⁹ This mimics numpy's behavior.

- `get_data` extracts a part of the data and returns it packed in a new *distributed_data_object*
- `set_data` modifies parts of the *distributed_data_object*'s data,
- `get_local_data` returns the process' local data,
- `set_local_data` directly modifies the process' local data.

In principle, one could mimic the behavior of `set_full_data` with `set_data` but the former is faster since there are no indexing checks involved. *distributed_data_objects* support large parts of numpy's indexing functionality, via the methods `get_data` and `set_data`.²⁰ This includes simple and advanced indexing, slicing and boolean extraction. Note that multidimensional advanced indexing is currently not supported by the slicing distributor: something like

```

1 | In [10]: a = np.arange(12).reshape(3, 4)
2 | In [11]: obj = distributed_data_object(a)
3 | In [12]: obj
4 | Out[12]: <distributed_data_object>
5 |          array([[ 0,  1,  2,  3],
6 |                [ 4,  5,  6,  7],
7 |                [ 8,  9, 10, 11]])
8 |
9 | # Simple indexing
10 | In [13]: obj[2,1]
11 | Out[13]: 9
12 |
13 | # Advanced indexing
14 | In [14]: index_tuple = (np.array([1, 1, 2, 2, 2, 2]),
15 |                        np.array([2, 3, 0, 1, 2, 3]))
16 | In [15]: obj[index_tuple]
17 |
18 | Out[15]: <distributed_data_object>
19 |          array([ 6,  7,  8,  9, 10, 11])
20 |
21 | # Slicing
22 | In [16]: obj[:, :-2]
23 | Out[16]: <distributed_data_object>
24 |          array([[ 3,  1],
25 |                [ 7,  5],
26 |                [11,  9]])
27 |
28 | # Boolean extraction
29 | In [17]: obj[obj>5]
30 | Out[17]: <distributed_data_object>
          array([ 6,  7,  8,  9, 10, 11])

```

will throw an exception.

²⁰ These are the methods getting called through Python's `obj[...] = ...` notation.

```

1 | In [18]: a = np.arange(12).reshape(3, 4)
2 | In [19]: obj = distributed_data_object(a)
3 | In [20]: obj[obj>5] = [11, 22, 33, 44, 55, 66]
4 | In [21]: obj
5 | Out[21]: <distributed_data_object>
6 |          array([[ 0,  1,  2,  3],
7 |                [ 4,  5, 11, 22],
8 |                [33, 44, 55, 66]])

```

All those indexing variants can also be used for setting array data, for example:

Allowed types for input data are scalars, tuples, lists, numpy ndarrays and *distributed_data_objects*. Internally the individual processes then extract the locally relevant portion of it.

As it is extremely costly, D2O tries to avoid inter-process communication whenever possible. Therefore, when using the `get_data` method the returned data portions remain on their processes. In case of a *distributed_data_object* with a slicing distribution strategy the *freemform distributor* is used for this, cf. "[Distribution strategies](#)" section.

Local keys

The distributed nature of D2O adds an additional degree of freedom when getting (setting) data from (to) a *distributed_data_object*. The indexing discussed in "[Getting and setting data](#)" section is based on the assumption that the involved key- and data-objects are the same for every MPI node. But in addition to that, D2O allows the user to specify node-individual keys and data. This, for example, can be useful when data stored as a *distributed_data_object* must be piped into a software module which needs very specific portions of the data on each MPI process. If one is able to describe those data portions as array-indexing keys—like slices—then the user can do this data redistribution within a single line. The following script—executed by two MPI processes—illustrates the point of local keys.

```

1 | # local_keys.py
2 | from mpi4py import MPI
3 | import numpy as np
4 | from d2o import distributed_data_object

```

```

5 rank = MPI.COMM_WORLD.rank
6
7 # initializing some data
8 obj = distributed_data_object(np.arange(16)*2)
9
10 print (rank, obj)
11
12 # getting data using the same slice on both processes
13 print (rank, obj.get_data(key=slice(None, None, 2)))
14
15 # getting data using different slices
16 print (rank, obj.get_data(key=slice(None, None, 2+rank),
17                             local_keys=True))
18
19 # getting data using different distributed_data_objects
20 key_tuple = (distributed_data_object([1, 3, 5, 7]),
21              distributed_data_object([2, 4, 6, 8]))
22 key = key_tuple[rank]
23 print (rank, obj.get_data(key=key, local_keys=True))

```

The first print statement shows the starting data: the even numbers ranging from 0 to

```

(0, <distributed_data_object>
  array([ 0,  2,  4,  6,  8, 10, 12, 14]))
(1, <distributed_data_object>
  array([16, 18, 20, 22, 24, 26, 28, 30]))

```

30: In line 13 we extract every second entry from `obj` using `slice(None, None, 2)`. Here, no inter-process communication is involved; the yielded data remains on the original node. The output of the print statement reads:

```

(0, <distributed_data_object>
  array([ 0,  4,  8, 12]))
(1, <distributed_data_object>
  array([16, 20, 24, 28]))

```

In line 17 the processes ask for different slices of the global data using the keyword `local_keys=True`: process 0 requests every second element whereas process 1 requests every third element from `obj`. Now communication is required to redistribute the data and the results are stored in the individual processes.

```

(0, <distributed_data_object>
  array([ 0,  4,  8, 12, 16, 20, 24, 28]))
(1, <distributed_data_object>
  array([ 0,  6, 12, 18, 24, 30]))

```

In line 23 we use *distributed_data_objects* as indexing objects. Process 0 requests the elements at positions 1, 3, 5 and 7; process 1 for those at 2, 4, 6 and 8. The peculiarity here is that the keys are not passed to `obj` as a complete set of local *distributed_data_object* instances. In fact, the processes only hand over their very local instance of the

keys. D2O is aware of this and uses the `d2o_librarian` in order to reassemble them, cf. "[The d2o librarian](#)" section. The output reads:

```
(0, <distributed_data_object>
  array([ 2,  6, 10, 14]))
(1, <distributed_data_object>
  array([ 4,  8, 12, 16]))
```

The `local_keys` keyword is also available for the `set_data` method. In this case the keys as well as the data updates will be considered local objects. The behaviour is analogous to the one of `get_data`: The individual processes store the locally relevant part of the `to_key` using their distinct `data[from_key]`.

The d2o librarian

A *distributed_data_object* as an abstract entity in fact consists of a set of Python objects that reside in memory of each MPI process. Global operations on a *distributed_data_object* necessitate that all those local instances of a *distributed_data_object* receive the same function calls; otherwise unpredictable behavior or a deadlock could happen. Let us discuss an illustrating example, the case of extracting a certain piece of data from a *distributed_data_object* using slices, cf. "[The d2o librarian](#)" section. Given a request for a slice of data, the MPI processes check which part of their data is covered by the slice, and build a new *distributed_data_object* from that. Thereby they communicate the size of their local data, maybe make sanity checks, and more. If this `get_data(slice(...))` function call is not made on every process of the cluster, a deadlock will occur as the 'called' processes wait for the 'uncalled' ones. However, especially when using the `local_keys` functionality described in "[Local keys](#)" section algorithmically one would like to work with different, i.e. node-individual *distributed_data_objects* at the same time. This raises the question: given only one local Python object instance, how could one make a global call on the complete *distributed_data_object* entity it belongs to? For this the `d2o_librarian` exists. During initialization every *distributed_data_object* registers itself with the `d2o_librarian` which returns a unique index. Later, this index can be used to assemble the full *distributed_data_object* from just a single local instance. The following code illustrates the workflow.

```
mpirun -n 4 python librarian.py
```

```

1 | # librarian.py
2 | from mpi4py import MPI
3 | import numpy as np
4 | from d2o import distIndex: 1          , d2o_librarian
5 |     (0, array([[2, 3]]))
6 | comm = MPI.COMM_WORLD(1, array([[6, 7]]))
7 | rank = comm.rank    (2, array([[10, 11]]))
8 |     (3, array([[14, 15]]))
9 | # initialize four diIndex: 2          _data_objects
10 | obj = distributed_da(0, array([[4, 6]]) (16).reshape((4,4))
11 | obj_list = (obj, 2*o(1, array([[12, 14]]))
12 |             (2, array([[20, 22]]))
13 | # every process gets (3, array([[28, 30]]))spective full array
14 | individual_object = Index: 3
15 | individual_index = i(0, array([[6, 9]]) dex
16 | index_list = comm.al(1, array([[18, 21]])index)
17 |             (2, array([[30, 33]]))
18 | for index in index_l(3, array([[42, 45]])
19 |     # resemble the cIndex: 4          node
20 |     current_object = (0, array([[ 8, 12]])x]
21 |     if rank == 0: pr(1, array([[24, 28]]) (index))
22 |     # take a slice o(2, array([[40, 44]])
23 |     print (rank, cur(3, array([[56, 60]]).get_local_data())

```

The output reads:

The D2O-librarian's core-component is a *weak dictionary* wherein *weak references* to the local *distributed_data_object* instances are stored. Its peculiarity is that those weak references do not prevent Python's garbage collector from deleting the object once no regular references to it are left. By this, the librarian can keep track of the *distributed_data_objects* without, at the same time, being a reason to hold them in memory.

Copy methods

D2O's array copy methods were designed to avoid as much Python overhead as possible. Nevertheless, there is a speed penalty compared to pure numpy arrays for a single process; cf. "[Performance and scalability](#)" section for details. This is important as binary operations like addition or multiplication of an array need a copy for returning the results. A special feature of D2O is that during a full copy one may change certain array properties such as the data type and the distribution strategy:

```

1 | In [22]: a = np.arange(4)
2 | In [23]: obj = distributed_data_object(a) # dtype == np.int
3 | In [24]: p = obj.copy(dtype=np.float,
4 |                       distribution_strategy='not')
5 | In [25]: (p.distribution_strategy, p)
6 | Out[25]: ('not', <distributed_data_object>
7 |           array([ 0.,  1.,  2.,  3.]))

```

When making empty copies one can also change the global or local shape:

```

1 | In [26]: obj = distributed_data_object(global_shape=(4,4),
2 |                                     dtype=np.float)
3 | # only the shape gets changed
4 | In [27]: obj.copy_empty(global_shape=(2,2))
5 | Out[27]: <distributed_data_object>
6 |          array([[ 6.90860823e-310,   9.88131292e-324],
7 |                 [ 9.88131292e-324,   1.97626258e-323]])

```

Fast iterators

A large class of problems requires iteration over the elements of an array one by one [27]. Whenever possible, Python uses special *iterators* for this in order to keep computational costs at a minimum. A toy example is

```

1 | In [28]: l = [9, 8, 7, 6]
2 | In [29]: for item in l:
3 |           print item
4 | .....:
5 | 9
6 | 8
7 | 7
8 | 6

```

Inside Python, the `for` loop requests an iterator object from the list `l`. Then the loop pulls elements from this iterator until it is exhausted. If an object is not able to return an iterator, the `for` loop will extract the elements using `__getitem__` over and over again. In the case of *distributed_data_objects* the latter would be extremely inefficient as every `__getitem__` call incorporates a significant amount of communication. In order to circumvent this, the iterators of *distributed_data_objects* communicate the process' data in chunks that are as big as possible. Thereby we exploit the knowledge that the array elements will be fetched one after another by the iterator. An examination of the performance difference is done in "Appendix 2".

Appendix 2: Iterator performance

As discussed in "Fast iterators" section, iterators are a standard tool in Python by which objects control their behavior in `for` loops and list comprehensions [27]. In order to speed up the iteration process, *distributed_data_objects* communicate their data as chunks chosen to be as big as possible. Thereby `D2O` builds upon the knowledge that elements will be fetched one after another by the iterator as long as further elements are requested.²¹ Additionally, by its custom iterator interface `D2O` avoids that the full data consolidation logic is invoked for every entry. Because of this, the performance gain is roughly a factor of 30 even for single-process scenarios as demonstrated in the following example:

²¹ This has the downside, that if the iteration was stopped prematurely, data has been communicated in vain.


```

1 In [1]: length = 1000
2 In [2]: obj = distributed_data_object(np.arange(length))
3
4 In [3]: def using_iterators(obj):
5         for i in obj:
6             pass
7
8 In [4]: def not_using_iterators(obj):
9         for j in xrange(length):
10            obj[j]
11
12 In [5]: %timeit not_using_iterators(obj)
13         10 loops, best of 3: 104 ms per loop
14
15 In [6]: %timeit using_iterators(obj)
16         100 loops, best of 3: 2.92 ms per loop

```

Received: 10 June 2016 Accepted: 30 August 2016

Published online: 15 September 2016

References

- Greiner M, Schnitzler DHFM, Ensslin TA. Tomography of the galactic free electron density with the square kilometer array. ArXiv e-prints. 2015. 1512.03480.
- Junklewitz H, Bell MR, Selig M, Enßlin TA. RESOLVE: a new algorithm for aperture synthesis imaging of extended emission in radio astronomy. *Astron Astrophys*. 2016;586:76. doi:10.1051/0004-6361/201323094. arxiv:1311.5282.
- Selig M, Bell MR, Junklewitz H, Oppermann N, Reinecke M, Greiner M, Pachajoa C, Enßlin TA. NIFTY— numerical information field theory. A versatile PYTHON library for signal inference. *Astron Astrophys*. 2013;554:26. doi:10.1051/0004-6361/201321236.
- van der Walt S, Colbert SC, Varoquaux G. The numpy array: a structure for efficient numerical computation. *Comput Sci Eng*. 2011;13(2):22–30. doi:10.1109/MCSE.2011.37.
- Forum MPI. MPI: a message passing interface standard. *Int J Supercomput Appl*. 1994;8(3/4):159–416.
- Message Passing Interface Forum. MPI2: a message passing interface standard. *High Perform Comput Appl*. 1998;12(1–2):1–299.
- Enthought I. DistArray: think globally, act locally. 2016. <http://docs.enthought.com/distarray/>. Accessed 24 Mar 2016.
- Frigo M. A fast fourier transform compiler. In: Proceedings of the ACM SIGPLAN 1999 conference on programming language design and implementation. PLDI '99. New York; 1999. doi:10.1145/301618.301661. <http://doi.acm.org/10.1145/301618.301661>.
- Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC. ScaLAPACK users' guide. Philadelphia: Society for Industrial and Applied Mathematics; 1997.
- Dadone A, Grossman B. Ghost-cell method for inviscid two-dimensional flows on cartesian grids. *AIAA J*. 2004;42:2499–507. doi:10.2514/1.697.
- Pérez F, Granger BE. IPython: a system for interactive scientific computing. *Comput Sci Eng*. 2007;9(3):21–9. doi:10.1109/MCSE.2007.53.
- Team S. ScaLAPACK web page. 2016. <http://www.netlib.org/scalapack/>. Accessed 23 Mar 2016.
- Balay S, Abhyankar S, Adams MF, Brown J, Brune P, Buschelman K, Dalcín L, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Rupp K, Smith BF, Zampini S, Zhang H. PETSc web page. 2015. <http://www.mcs.anl.gov/petsc>.
- McKerns MM, Strand L, Sullivan T, Fang A, Aivazis MAG. Building a framework for predictive science. *CoRR*. 2012. arXiv:1202.1056.
- Strohmaier E, Dongarra J, Simon H, Meuer M. The TOP500 project. 2015. <http://www.top500.org/lists/2015/11/>. Accessed 24 Mar 2016.
- Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. In: Proceedings of the 2Nd USENIX conference on hot topics in cloud computing. HotCloud'10. Berkeley: USENIX Association. Berkeley; 2010. p. 10–10. <http://dl.acm.org/citation.cfm?id=1863103.1863113>.
- Apache Software Foundation. Hadoop. 2016. <https://hadoop.apache.org>. Accessed 23 Mar 2016.
- Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, Castain RH, Daniel DJ, Graham RL, Woodall TS. Open MPI: goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI users' group meeting. Budapest; 2004. p. 97–104.
- Team M. MPICH2: high-performance portable MPI. 2016. <http://www.mcs.anl.gov/mpich2>. Accessed 24 Mar 2016.
- Corporation I. Intel MPI library. 2016. <https://software.intel.com/en-us/intel-mpi-library>. Accessed 6 June 2016.
- Dalcín L, Paz R, Storti M. MPI for python. *J Parallel Distrib Comput*. 2005;65(9):1108–15. doi:10.1016/j.jpdc.2005.03.010.

22. Gomersall H. pyFFTW: a pythonic wrapper around FFTW. We use the mpi branch available at <https://github.com/fredRos/pyFFTW>. 2016. <https://hgomersall.github.io/pyFFTW>. Accessed 23 Mar 2016.
23. Universe EC. Excellence cluster universe. 2016. <http://www.universe-cluster.de/c2pap>. Accessed 6 Apr 2016.
24. Enßlin TA, Frommert M, Kitaura FS. Information field theory for cosmological perturbation reconstruction and non-linear signal analysis. *Phys Rev D*. 2009;80:105005. doi:10.1103/PhysRevD.80.105005. arxiv: 0806.3474.
25. Wiener N. Extrapolation, interpolation and smoothing of stationary time series, with engineering applications. In: note: Originally issued in Feb 1942 as a classified Nat. Council Rep: Defense Res. New York: Technology Press and Wiley; 1949.
26. Frigo M, Johnson SG. The design and implementation of FFTW3. In: Proceedings of the IEEE, Vol 93(2). Special issue on "Program Generation, Optimization, and Platform Adaptation". 2005. p. 216–231
27. Ka-Ping Yee GVR. PEP 234—Iterators. 2016. <https://www.python.org/dev/peps/pep-0234/>. Accessed 12 Apr 2016.

Submit your manuscript to a SpringerOpen[®] journal and benefit from:

- ▶ Convenient online submission
- ▶ Rigorous peer review
- ▶ Immediate publication on acceptance
- ▶ Open access: articles freely available online
- ▶ High visibility within the field
- ▶ Retaining the copyright to your article

Submit your next manuscript at ▶ springeropen.com
