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

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

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

1 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 \cite{1} or multi-frequency imaging in radio astronomy \cite{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)\cite{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.

2 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 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 [5]: “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 [6] may rely on a different data-distribution model than a package for linear algebra operations like ScaLAPACK [7][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 not only storing a piece of a global array, but also its neighboring (ghost) cells [8].

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. Referring to the examples given in section 1, we aim at running a code at double resolution rather than reducing its run time by a factor of two, i.e. weak scaling, not strong scaling.

3 Choosing the Right Level of Parallelization
D2O distributes numerical arrays over a cluster in order to parallelize and therefore 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. section 5, 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.

[1] FFTW distributes slices of data, while ScaLAPACK uses a block-cyclic distribution pattern.
3.1 Vertical & 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 need not 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.

3.2 High- & 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* \(^9\) 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.

3.3 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\(^5\), binary\(^6\) 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

---

\(^2\)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 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.
we want to tackle involve at least $10^8$ elements but most of the TOP500 [10] supercomputers possess $10^6$ cores or 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.

3.4 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. With the potential scaling problems mentioned above in mind, we repeat our primary objective. The goal is not to reduce the run time of already doable computations but rather to render extremely costly ones possible in the first place. 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 experienced a good scaling behavior with up to $\approx 10^3$ processes[7] for problems of size $8192^2$, cf. section 9. Hence for our applications the advantages of low-level parallelization clearly outweigh its drawbacks.

3.5 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.

[7] This was the maximum number of processes which was available for testing
4 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 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 [11] and hadoop [12]: These modern frameworks are very powerful but regrettable 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 [13, 14]: The Message Passing Interface is available on most HPC clusters via well-tested implementations like OpenMPI [15], MPICH2 [16] or Intel MPI [17]. A Python interface is given by the Python module mpi4py [18]. MPI furthermore offers the right level of abstraction for hands-on control of distribution strategies.

Given these features we decide to use MPI as parallelization environment for d2o.

5 Alternative Packages

There are several alternatives to d2o. We discuss the differences to d2o and why the alternatives are not sufficient for our needs.

5.1 DistArray

DistArray [5] 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 [19] 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. section 2. Nevertheless, its theme also applies to d2o: “Think globally, act locally”.

5.2 scalapy (ScalAPACK)

scalapy is a Python wrapper around ScalAPACK [7], which is “a library of high-performance linear algebra routines for parallel distributed memory machines” [20]. 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. However, 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.

5.3 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” [21]. 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.

6 Internal Structure
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. The distributed_data_object itself constitutes the 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, albeit the distributed_data_object itself will never use or act on this attribute directly. This follows from the fact that by design it must not make any assumptions about the specific object which is stored in “data”.

For all tasks that require knowledge about the certain distribution strategy every distributed_data_object possesses an instance of the d2o.distributor class. 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 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. 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 of 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

[8] The slicing is done along the first array axis.
[9] The fftw-distributor uses routines from the pyFFTW [22, 6] package [6] 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.
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.

7 Basic Usage

In the subsequent sections we will illustrate the usage of D2O in order to explain its functionality and behavior. 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](https://gitlab.mpcdf.mpg.de/ift/D2O).

7.1 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.

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

Now we set up some test data using numpy.

```
In [3]: a = np.arange(12).reshape((3, 4))
In [4]: a
```

```
Out[4]: array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]])
```

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

```
In [5]: obj = distributed_data_object(a)
In [6]: obj
```

```
< distributed_data_object >
   array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]])
```

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

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

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

```python
In [8]: a + 1; # calls a.\_\_add\_(1) -> returns a numpy array
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.

```python
In [10]: a + obj # numpy converts obj to a ndarray -> slow
Out[10]: array([[ 0, 2, 4, 6],
             [ 8, 10, 12, 14],
             [16, 18, 20, 22]])

In [11]: obj + a # obj processes a -> fast
Out[11]: < distributed_data_object >
    array([[ 0, 2, 4, 6],
            [ 8, 10, 12, 14],
            [16, 18, 20, 22]])
```
7.3 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. 8.3.

```python
In [12]: obj
Out[12]: <distributed_data_object>
        array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])

In [13]: obj[1]  # extract a row
Out[13]: <distributed_data_object>
        array([4 ,  5,  6,  7])

In [14]: obj[1,-2]  # extract single entry
Out[14]: 6

In [15]: obj[::2, 1::2]  # slicing notation
Out[15]: <distributed_data_object>
        array([[1, 3],
                [9, 11]])

# sets data with slicing
In [16]: obj[::2, 1::2] = [[111, 222], [333, 444]]
In [17]: obj
Out[17]: <distributed_data_object>
        array([[ 0, 111,  2, 222],
                [ 4,  5,  6,  7],
                [ 8,  333, 10, 444]])

By default it is assumed that all processes use the same key-object when accessing data. See section 8.4 for more details regarding process-individual indexing.

7.4 Distribution Strategies

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

```python
In [18]: obj = distributed_data_object(a, distribution_strategy='equal')
In [19]: obj.distribution_strategy
Out[19]: 'equal'
```

See section 8.1 for more information on distribution strategies.

7.5 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:

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

```
(0, array([[1, 2]]))
(1, array([], shape=(0, 2), dtype=int64)) # empty
(2, array([[9, 10, 11]]))
(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

\(^{[12]}\)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.
we store a small 2x2 block \( b \) in the lower middle of \( \text{obj} \). The process’ local data reads:

\[
(0, \text{array([[ 0, 1, 2, 3]])}) \\
(1, \text{array([[ 4, 5, 6, 7]])}) \\
(2, \text{array([[ 8, 0, -1, 11]])}) \\
(3, \text{array([[12, -2, -3, 15]])})
\]

Finally, in line 25 we use \text{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.

\[
(0, \text{array([[ 0, 1, 2, 3],} \\
\quad \text{[ 4, 5, 6, 7],} \\
\quad \text{[ 8, 0, -1, 11],} \\
\quad \text{[12, -2, -3, 15]])})
\]

**8 Advanced Usage and Functional Behavior**

In this section we discuss specifics regarding the design and functional behavior of \text{d2o}. We set things up by importing \text{numpy} and \text{d2o.distributed_data_object}:

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

**8.1 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 \text{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.

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

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

\[
(0, \text{\textquotesingle\textquotesingle not\textquotesingle\textquotesingle, \text{array}([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]))\\
(1, \text{\textquotesingle\textquotesingle not\textquotesingle\textquotesingle, \text{array}([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]))\\
(2, \text{\textquotesingle\textquotesingle not\textquotesingle\textquotesingle, \text{array}([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]))
\]

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 \((2, 1, 1)\):

\[
(0, \text{\textquotesingle\textquotesingle equal\textquotesingle\textquotesingle, \text{array}([[0, 1, 2, 3], [4, 5, 6, 7]]))\\
(1, \text{\textquotesingle\textquotesingle equal\textquotesingle\textquotesingle, \text{array}([[8, 9, 10, 11]]))\\
(2, \text{\textquotesingle\textquotesingle equal\textquotesingle\textquotesingle, \text{array}([[12, 13, 14, 15]]))
\]

The “fftw” distribution strategy is very similar to “equal” but uses functions from FFTW\([6]\). 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 \((2, 2, 0)\):

\[
(0, \text{\textquotesingle\textquotesingle fftw\textquotesingle\textquotesingle, \text{array}([[0, 1, 2, 3], [4, 5, 6, 7]]))\\
(1, \text{\textquotesingle\textquotesingle fftw\textquotesingle\textquotesingle, \text{array}([[8, 9, 10, 11], [12, 13, 14, 15]]))\\
(2, \text{\textquotesingle\textquotesingle fftw\textquotesingle\textquotesingle, \text{array}([], \text{shape}=(0, 4), \text{dtype}=\text{int64}))
\]

A “freeform” array is built from a process-local perspective: each process gets its individual local data. In our example, we use \texttt{a\textasciitilde rank} as the local data arrays – each being of shape \((4, 4)\) – during the initialization of the \texttt{distributed.data.object}. By this, a global shape of \((12, 4)\) is produced. The local data reads:

\[
(0, \text{\textquotesingle\textquotesingle freeform\textquotesingle\textquotesingle, \text{array}([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]))\\
(1, \text{\textquotesingle\textquotesingle freeform\textquotesingle\textquotesingle, \text{array}([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]]))
\]
This yields a global shape of (12, 4). In order to consolidate the data the method
`obj.get_full_data()` is used, cf. section 8.3.

8.2 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:

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

The initial data is interpreted as global data. The default distribution strategy[13]
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:

```python
In [6]: obj = distributed_data_object(global_data=a,
                                        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:

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

[13] Depending on whether `pyfftw` is available or not, the `equal`- or the `fftw`-distribution strategy is
If the data type is specified neither implicitly by some initial data nor explicitly via `dtype`, `distributed_data_object` uses `float` as a default\[^{14}\]. 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. section 8.1:

```python
In [9]: obj = distributed_data_object(
    local_data=a,
    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.

### 8.3 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,
- `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`\[^{15}\]. 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

```python
obj[np.array([[1,2], [0,1]]), np.array([[0,1], [2,3]])]
```

will throw an exception.

```python
In [10]: a = np.arange(12).reshape(3, 4)
```

---

\[^{14}\]: This mimics numpy’s behavior.

\[^{15}\]: These are the methods getting called through Python’s `obj[...]` notation.
In [11]: obj = distributed_data_object(a)
In [12]: obj
Out[12]: <distributed_data_object> 
    array([[ 0,  1,  2,  3],  
            [ 4,  5,  6,  7],  
            [ 8,  9, 10, 11]])

# Simple indexing
In [13]: obj[2,1]
Out[13]: 9

# Advanced indexing
In [14]: index_tuple = (np.array([1, 1, 2, 2, 2, 2]), np.array([2, 3, 0, 1, 2, 3]))
In [15]: obj[index_tuple]
Out[15]: <distributed_data_object> 
    array([ 6,  7,  8,  9, 10, 11])

# Slicing
In [16]: obj[:, ::-2]
Out[16]: <distributed_data_object> 
    array([[ 3,  1],  
            [ 7,  5],  
            [11,  9]])

# Boolean extraction
In [17]: obj[obj > 5]
Out[17]: <distributed_data_object> 
    array([ 6,  7,  8,  9, 10, 11])

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

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

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 freeform distributor is used for this, cf. section 8.1.

8.4 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 section 8.3 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.

```python
mpirun -n 2 python local_keys.py
```

```python
# local_keys.py
from mpi4py import MPI
import numpy as np
from d2o import distributed_data_object

rank = MPI.COMM_WORLD.rank

# initializing some data
obj = distributed_data_object(np.arange(16)**2)

print(rank, obj)

# getting data using the same slice on both processes
print(rank, obj.get_data(key=slice(None, None, 2)))

# getting data using different slices
print(rank, obj.get_data(key=slice(None, None, 2+rank), local_keys=True))

# getting data using different distributed_data_objects
key_tuple = (distributed_data_object([1, 3, 5, 7]), distributed_data_object([2, 4, 6, 8]))
key = key_tuple[rank]
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 30:

(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]))

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. section 8.5. 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].

8.5 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. section 8.3. 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 section 8.4 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

```python
# librarian.py
from mpi4py import MPI
import numpy as np
from d2o import distributed_data_object, d2o_librarian

comm = MPI.COMM_WORLD
rank = comm.rank
```
# initialize four different distributed_data_objects

```
obj = distributed_data_object(np.arange(16).reshape((4,4)))
obj_list = (obj, 2*obj, 3*obj, 4*obj)
```

# every process gets its part of the respective full array

```
individual_object = obj_list[rank]
individual_index = individual_object.index
index_list = comm.allgather(individual_index)
```

```
for index in index_list:
    # resemble the current d2o on every node
    current_object = d2o_librarian[index]
    if rank == 0: print('Index : ' + str(index))
    # take a slice of data
    print(rank, current_object[:, 2:4].get_local_data())
```

The output reads:

Index: 1
(0, array([[2, 3]]))
(1, array([[6, 7]]))
(2, array([[10, 11]]))
(3, array([[14, 15]]))
Index: 2
(0, array([[4, 6]]))
(1, array([[12, 14]]))
(2, array([[20, 22]]))
(3, array([[28, 30]]))
Index: 3
(0, array([[6, 9]]))
(1, array([[18, 21]]))
(2, array([[30, 33]]))
(3, array([[42, 45]]))
Index: 4
(0, array([[8, 12]]))
(1, array([[24, 28]]))
(2, array([[40, 44]]))
(3, array([[56, 60]]))

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.

8.6 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. section 9 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:
In [22]: a = np.arange(4)
In [23]: obj = distributed_data_object(a)  # dtype == np.int
In [24]: p = obj.copy(dtype=np.float,
   distribution_strategy='not')
In [25]: (p.distribution_strategy, p)
Out[25]: ('not', <distributed_data_object>
   array([ 0., 1., 2., 3.]))

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

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

8.7 Fast Iterators
A large class of problems requires iteration over the elements of an array one by one [23]. Whenever possible, Python uses special iterators for this in order to keep computational costs at a minimum. A toy example is

In [28]: l = [9, 8, 7, 6]
In [29]: for item in l:
   print item
....:
9
8
7
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 section 9.3.

9 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 [24][16]. 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

[16] The C2PAP computing cluster consists of 128 nodes, possessing two Intel Xeon CPU E5-2680 (8 cores 2.70GHz + hyper-threading) and 64 GiB RAM each. The nodes are connected via Mellanox Infiniband 40 Gbits/s
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.

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 d2o’s management overhead becomes negligible compared to the purely numerical costs. This size corresponds to a two-dimensional grid with a resolution of $256 \times 256$ or equivalently 0.5 MiB of 64-bit doubles. In section 9.1 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 parallel regime. In section 9.2, the scaling analysis regarding the MPI process count is done with an array size for which the local-process overhead is negligible compared to the local numerical costs. This is because we are interested in the costs arising from inter-process communication compared to those of actual numerics.

### 9.1 Scaling with the Array Size

One may think of D2o as a layer of abstraction which 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\[17\]. Hence, there the speed of object creation can be crucial for overall performance. 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. 8.5), and, if activated, inter-MPI-process communication must be done for performing sanity checks on the user input.

By default the initialization requires 60 µs 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\[17\]. Exceptions to this are inplace operations which reuse the input array for the output data.
tasks. Hence, those 60 µs represent d2o’s constant overhead compared to numpy, since a comparable numpy array requires ≈ 0.4 µs 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 µs from 60 µs to 33 µs.

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 µs and 4.3 µs, respectively.

Table 2 shows the performance ratio in percent between serial d2o and numpy. The array sizes range from $2^0 = 1$ to $2^{25} \approx 3.3 \cdot 10^7$. In the table, 100% would mean that d2o was as fast as numpy.

The previous section already suggested that for tasks which primarily consist of initialization work – like array creation or copy_empty – d2o will clearly lie 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. Still, with a considerably large array size, d2o’s efficiency becomes comparable to that of numpy.

Similarly to obj[::-2], the remaining functions in the table return a d2o 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
218 elements need 256 KiB of space which is congruent with the amount of 256 KiB L2 cache C2PAP’s Intel E5-2680 CPUs posses. d2o’s memory overhead is now responsible for the fact, that it cannot profit from the L2 cache anymore, whereas the numpy array still operates fast. Once the array size is above 222 elements numpy’s just as d2o’s array-object are too large for profiting from the L2 cache and therefore become comparably fast: the relative performance is then greater than 98%.

Thus, d2o is ideally used for arrays larger than 216 = 65536 elements which corresponds to 64 KiB. From there the management overhead becomes less significant than the actual numerical costs.

9.2 Scaling with the Number of Processes

Now we analyze the scaling behavior of d2o when run with several MPI processes. Repeating the beginning of section 9, there are three contributions to the execution time. First, fixed management overhead that every process has to cover itself. Second, 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 3 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, this ratio would be equal to 1.

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. As more processors imply more CPU cache, we even encounter super-linear scaling.

Comparing sum(axis=0) with sum(axis=1) illustrates the performance difference between those operations that involve 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 3 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.

9.3 Iterators

As discussed in section 8.7, iterators are a standard tool in Python by which objects control their behavior in for loops and list comprehensions [23]. 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.\textsuperscript{[18]} Additionally, by its custom iterator interface \texttt{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:

```python
In [1]: length = 1000
In [2]: obj = distributed_data_object(np.arange(length))
In [3]: def using_iterators(obj):
    ...:     for i in obj:
    ...:         pass
In [4]: def not_using_iterators(obj):
    ...:     for j in xrange(length):
    ...:         obj[j]
In [5]: %timeit not_using_iterators(obj)
    ...:     10 loops, best of 3: 104 ms per loop
In [6]: %timeit using_iterators(obj)
    ...:     100 loops, best of 3: 2.92 ms per loop
```

9.4 Real-World Application Speedup: Wiener filter

\texttt{d2o} was initially developed for \textsc{NIFTy}\textsuperscript{[3]}, a library for building signal inference algorithms in the framework of information field theory (IFT)\textsuperscript{[25]}. Within \textsc{NIFTy} v2 all of the parallelization is done via \texttt{d2o}; the code of \textsc{NIFTy} itself is almost completely parallelization and completely MPI agnostic.

A standard computational operation in IFT-based signal reconstruction is the Wiener filter\textsuperscript{[26]}. For this performance test, we use a Wiener filter implemented in \textsc{NIFTy} to reconstruct the realization of a Gaussian random field – called the \textit{signal}. Assume we performed a hypothetical measurement which produced some \textit{data}. The data model is

\begin{equation}
data = R(signal) + noise
data = R(signal) + noise
data = R(signal) + noise
\end{equation}

where \textit{R} is a smoothing operator and \textit{noise} is additive Gaussian noise. In figure 1 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 \textit{data}.

Table 1 shows the scaling behavior of the reconstruction code, run with a resolution of 8192 x 8192. Here, \textit{n} is the number of used processes and \textit{t}_{\text{run}} the respective execution time. The relative speedup \textit{s}_{n} = t_{1}/t_{n} is the ratio of execution times: parallel versus serial. In the case of exactly linear scaling \textit{s}_{n} is equal to \textit{n}. Furthermore we define the scaling quality \textit{q} = 1/(1 + \log(n/s_{n})), which compares \textit{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.

| #nodes | 1 | 1 | 1 | 1 | 2 | 4 | 8 | 16 | 32 |
| #processes | 1 | 2 | 3 | 4 | 8 | 16 | 32 | 64 | 128 |
| $t_n$ | 1618 | 622.0 | 404.2 | 364.2 | 181.7 | 94.50 | 46.79 | 18.74 | 8.560 |
| $s_n = \frac{t_1}{t_n}$ | 1.00 | 2.60 | 4.00 | 4.44 | 8.90 | 17.1 | 34.6 | 86.3 | 189 |
| $q_n = 1/(1+\log(\frac{n}{s_n}))$ | 1.00 | 1.13 | 1.14 | 1.05 | 1.05 | 1.03 | 1.03 | 1.15 | 1.20 |

This benchmark illustrates that even in a real-life application super-linear scaling is possible to achieve. This is due to the operations that are needed in order to perform the Wiener filtering: basic point-wise arithmetics that do not involve any inter-process communication and Fourier transformations that are handled by the high-performance library FFTW\cite{27}. While the problem size remains constant, the amount of available CPU cache increases with the number of used processes. This is the most probable explanation for the observed super-linear scaling.

10 Summary & 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: we want to scale up the size of the numerical problems we solve, rather than to scale down the computing time of already doable problems. 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, and and last 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.

Competing interests
The authors declare that they have no competing interests.

Author’s 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.

Acknowledgments
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”. 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).
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.

References

1. Greiner, M., Schnitzeler, D.H.F.M., Ensslin, T.A.: Tomography of the Galactic free electron density with the Square Kilometer Array. ArXiv e-prints (2015). 1512.03480
2. Junklewitz, H., Bell, M.R., Selig, M., Enßlin, T.A.: RESOLVE: A new algorithm for aperture synthesis imaging of extended emission in radio astronomy. A&A 586, 76 (2016). doi:10.1051/0004-6361/201523094. 1511.5282
3. Selig, M., Bell, M.R., Junklewitz, H., Oppermann, N., Reinke, M., Greiner, M., Pachajoa, C., Enßlin, T.A.: NIFTY - Numerical Information Field Theory. A versatile PYTHON library for signal inference. A&A 554, 20 (2013). doi:10.1051/0004-6361/201321236. 1301.4409
4. van der Walt, S., Colbert, S.C., Varoquaux, G.: The numpy array: A structure for efficient numerical computation. Computing in Science and Engineering 13(2), 22–30 (2011). doi:10.1109/MCSE.2011.37
5. Enthought, I.: DistArray: Think globally, act locally (2016). http://docs.enthought.com/distarray/ Accessed 2016-03-24
6. Frigo, M.: A fast fourier transform compiler. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation. PLDI '99, pp. 169–180. ACM, New York, NY, USA (1999). doi:10.1145/301618.301661. http://doi.acm.org/10.1145/301618.301661
7. Blckford, L.S., Choi, J., Clevy, A., D’Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK Users’ Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA (1997)
8. Dadone, A., Grossman, B.: Ghost-Cell Method for Inviscid Two-Dimensional Flows on Cartesian Grids. AIAA Journal 42, 2499–2507 (2004). doi:10.2514/1.697
9. McKerns, M.M., Strand, L., Sullivan, T., Fang, A., Alavizz, M.A.G.: Building a framework for predictive science. CoRR abs/1202.1056 (2012)
10. Strohmaier, E., Dongarra, J., Simon, H., Meuer, M.: The TOP500 project (2015). http://www.top500.org/lists/2015/11/ Accessed 2016-03-24
11. Zaharia, M., Chowdhury, M., Franklin, M.J., 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, pp. 10–10. USENIX Association, Berkeley, CA, USA (2010). http://dl.acm.org/citation.cfm?id=1863103.1863113
12. Apache Software Foundation: Hadoop (2016). https://hadoop.apache.org Accessed 2016-03-23
13. Forum, M.P.I.: MPI: A message passing interface standard. International Journal of Supercomputer Applications 8(3/4), 159–416 (1994)
14. Message Passing Interface Forum: MPI2: A message passing interface standard. High Performance Computing Applications 12(1–2), 1–299 (1998)
15. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squires, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users’ Group Meeting, Budapest, Hungary, pp. 97–104 (2004)
16. Team, M.: MPICH2: High-Performance Portable MPI (2016). www.mcs.anl.gov/mpich2 Accessed 2016-03-24
17. Corporation, I.: Intel MPI Library (2016). http://www.mcs.anl.gov/petsc Accessed 2016-03-23
18. Dalcín, L., Paz, R., Storti, M.: [MPI] for python. Journal of Parallel and Distributed Computing 65(9), 1108–1115 (2005). doi:10.1016/j.jpdc.2005.03.010
19. Pérez, F., Granger, B.E.: IPython: a system for interactive scientific computing. Computing in Science and Engineering 9(3), 21–29 (2007). doi:10.1109/MCSE.2007.53
20. Team, S.: ScaLAPACK Web Page (2016). www.netlib.org/scalapack/ Accessed 2016-03-23
21. Balay, S., Abhyankar, S., Adams, M.F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Rupp, K., Smith, B.F., Zampini, S., Zhang, H.: PETSc Web page. http://www.mcs.anl.gov/petsc (2015). http://www.mcs.anl.gov/petsc
22. Gomersall, H.: pyFFTW: a pythonic wrapper around FFTW (2016). https://hgomersall.github.io/pyFFTW Accessed 2016-03-23
23. Ka-Ping Yee, G.v.R.: PEP 234 – Iterators (2016). https://www.python.org/dev/peps/pep-0234/ Accessed 2016-04-12
24. Universe, E.C.: Excellence Cluster Universe (2016). http://www.universe-cluster.de/c2pap Accessed 2016-04-06
25. Ensslin, T.A., Frommert, M., Kitaura, F.S.: Information field theory for cosmological perturbation reconstruction and non-linear signal analysis. Phys. Rev. D80, 105005 (2009). doi:10.1103/PhysRevD.80.105005. 0809.3474
26. Wiener, N.: Extrapolation, Interpolation and Smoothing of Stationary Time Series, with Engineering Applications. Technology Press and Wiley, New York (1949). note: Originally issued in Feb 1942 as a classified Nat. Defense Res. Council Rep.
27. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE 93(2), 216–231 (2005). Special issue on “Program Generation, Optimization, and Platform Adaptation"
Figures

Figure 1 Wiener filter reconstruction

Top left: True signal to be reconstructed. Top right: Data which is the result of smoothing the signal and adding Gaussian noise. Bottom: reconstructed maximum of the posterior (conditional) probability density function for the signal given the data $P(\text{signal}|\text{data})$. 
Tables

Table 2  
D2O’s relative performance to Numpy when scaling the array size. “100%” would correspond 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. Please see section 9.1 for discussion.

| array size | 2^0 | 2^2 | 2^4 | 2^6 | 2^8 | 2^10 | 2^12 | 2^14 | 2^16 | 2^18 | 2^20 | 2^22 | 2^24 | 2^25 |
|------------|-----|-----|-----|-----|-----|------|------|------|------|------|------|------|------|------|
| initialization | 0.6% | 0.6% | 0.7% | 0.7% | 0.7% | 0.7% | 0.7% | 0.7% | 0.8% | 0.7% | 0.8% | 0.8% | 5.4% | 5.6% |
| copy_empty | 3.8% | 3.9% | 4.0% | 4.1% | 4.6% | 3.9% | 3.9% | 4.0% | 4.0% | 4.1% | 4.2% | 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] | 12.2% | 12.2% | 12.2% | 12.2% | 13.1% | 1.5% | 2.1% | 4.8% | 15.0% | 22.2% | 28.7% | 41.1% | 45.8% | 47.3% | 47.7% |
| copy | 8.2% | 8.9% | 9.3% | 9.7% | 9.7% | 10.4% | 14.8% | 26.7% | 65.8% | 95.5% | 98.7% | 99.1% | 99.9% | 99.4% | 98.4% |
| obj + 0 | 6.6% | 6.5% | 6.7% | 7.1% | 7.1% | 9.5% | 16.3% | 35.0% | 65.2% | 34.7% | 44.7% | 98.7% | 99.7% | 96.5% | 100.0% |
| obj + obj | 3.1% | 3.1% | 3.2% | 3.6% | 4.0% | 5.7% | 13.3% | 31.9% | 64.0% | 34.6% | 45.0% | 98.7% | 99.9% | 99.5% | 99.8% |
| obj += obj | 5.2% | 5.3% | 5.4% | 6.0% | 6.7% | 11.3% | 22.8% | 46.4% | 75.3% | 91.9% | 97.8% | 98.0% | 99.8% | 99.3% | 99.7% |
| sqrt | 3.2% | 3.2% | 3.3% | 3.5% | 4.4% | 7.5% | 18.7% | 45.6% | 75.8% | 65.2% | 83.2% | 98.8% | 99.4% | 99.6% | 99.8% |
| bincount | 3.6% | 3.3% | 3.8% | 4.0% | 5.0% | 7.5% | 16.5% | 35.4% | 58.0% | 75.1% | 78.0% | 76.8% | 78.9% | 82.1% | 83.2% |

Table 3  
D2O’s relative performance to the single-process case when increasing both, the number of processes and the global array size proportionally. The arrays used for this tests had the global shape (n x 2048, 2048) with n being the number of processes. “100%” would correspond to the case were the speedup is equal to the number of processes. In order to guide the eye, values < 30% are printed italic, values ≥ 90% are printed bold. Please see section 9.2 for discussion.

| process count | 1 | 2 | 3 | 4 | 8 | 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=00) | 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% | 42.6% | 33.6% | 25.3% | 13.6% | 17.3% |
| copy | 100.0% | 104.7% | 103.1% | 101.3% | 103.3% | 105.3% | 104.4% | 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% | 99.9% | 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% | 98.6% | 98.0% |
| bincount | 100.0% | 103.0% | 101.2% | 99.9% | 98.8% | 97.6% | 94.1% | 88.3% | 79.4% | 65.8% |