mpi4py-fft Documentation

Release 2.0.5

Mikael Mortensen and Lisandro Dalcin

Aug 10, 2023

CONTENTS:

CHAPTER

ONE

MPI4PY-FFT

mpi4py-fft is a Python package for computing Fast Fourier Transforms (FFTs). Large arrays are distributed and communications are handled under the hood by MPI for Python (mpi4py). To distribute large arrays we are using a new and completely generic algorithm that allows for any index set of a multidimensional array to be distributed. We can distribute just one index (a slab decomposition), two index sets (pencil decomposition) or even more for higher-dimensional arrays.

In mpi4py-fft there is also included a Python interface to the FFTW library. This interface can be used without MPI, much like pyfftw, and even for real-to-real transforms, like discrete cosine or sine transforms.

1.1 Introduction

The Python package mpi4py-fft is a tool primarily for working with Fast Fourier Transforms (FFTs) of (large) multidimensional arrays. There is really no limit as to how large the arrays can be, just as long as there is sufficient computing powers available. Also, there are no limits as to how transforms can be configured. Just about any combination of transforms from the FFTW library is supported. Finally, mpi4py-fft can also be used simply to distribute and redistribute large multidimensional arrays with MPI, without any transforms at all.

The main contribution of mpi4py-fft can be found in just a few classes in the main modules:

- `mpifft`
- `pencil`
- `distarray`
- `libfft`
- `fftw`

The `mpifft.PFFT` class is the major entry point for most users. It is a highly configurable class, which under the hood distributes large dataarrays and performs any type of transform, along any axes of a multidimensional array.

The `pencil` module is responsible for global redistributions through MPI. However, this module is rarely used on its own, unless one simply needs to do global redistributions without any transforms at all. The `pencil` module is used heavily by the `PFFT` class.

The `distarray` module contains classes for simply distributing multidimensional arrays, with no regards to transforms. The distributed arrays created from the classes here can very well be used in any MPI application that requires a large multidimensional distributed array.

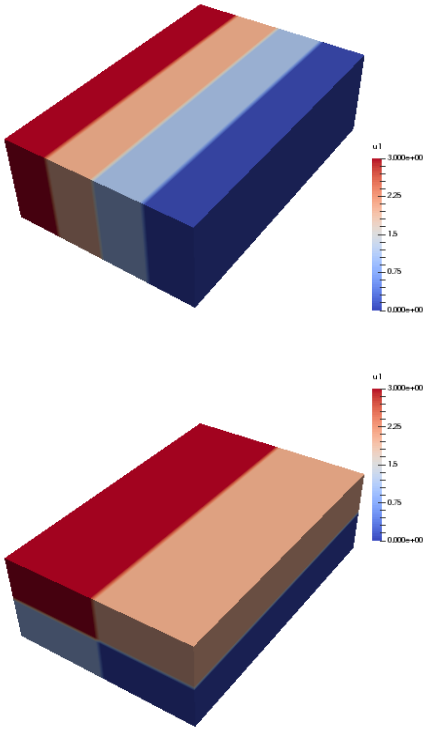
The `libfft` module provides a common interface to any of the serial transforms in the FFTW library.

The `fftw` module contains wrappers to the transforms provided by the FFTW library. We provide our own wrappers mainly because pyfftw does not include support for real-to-real transforms. Through the interface in `fftw` we can do here, in Python, pretty much everything that you can do in the original FFTW library.

1.2 Global Redistributions

In high performance computing large multidimensional arrays are often distributed and shared amongst a large number of different processors. Consider a large three-dimensional array of double (64 bit) precision and global shape $(512, 1024, 2048)$. To lift this array into RAM requires 8 GB of memory, which may be too large for a single, non-distributed machine. If, however, you have access to a distributed architecture, you can split the array up and share it between, e.g., four CPUs (most supercomputers have either 2 or 4 GB of memory per CPU), which will only need to hold 2 GBs of the global array each. Moreover, many algorithms with varying degrees of locality can take advantage of the distributed nature of the array to compute local array pieces concurrently, effectively exploiting multiple processor resources.

There are several ways of distributing a large multidimensional array. Two such distributions for our three-dimensional global array (using 4 processors) are shown below



Here each color represents one of the processors. We note that in the first image only one of the three axes is distributed, whereas in the second two axes are distributed. The first configuration corresponds to a slab, whereas the second corresponds to a pencil distribution. With either distribution only one quarter of the large, global array needs to be kept in rapid (RAM) memory for each processor, which is great. However, some operations may then require data that is not available locally in its quarter of the total array. If that is so, the processors will need to communicate with each other and send the necessary data where it is needed. There are many such MPI routines designed for sending and receiving data.

We are generally interested in algorithms, like the FFT, that work on the global array, along one axis at the time. To

be able to execute such algorithms, we need to make sure that the local arrays have access to all of its data along this axis. For the figure above, the slab distribution gives each processor data that is fully available along two axes, whereas the pencil distribution only has data fully available along one axis. Rearranging data, such that it becomes aligned in a different direction, is usually termed a global redistribution, or a global transpose operation. Note that with mpi4py-fft we always require that at least one axis of a multidimensional array remains aligned (non-distributed).

Distribution and global redistribution is in mpi4py-fft handled by three classes in the `pencil` module:

- `Pencil`
- `Subcomm`
- `Transfer`

These classes are the low-level backbone of the higher-level `PFFT` and `DistArray` classes. To use these low-level classes directly is not recommended and usually not necessary. However, for clarity we start by describing how these low-level classes work together.

Lets first consider a 2D dataarray of global shape (8, 8) that will be distributed along axis 0. With a high level API we could then simply do:

```
import numpy as np
from mpi4py_fft import DistArray
N = (8, 8)
a = DistArray(N, [0, 1])
```

where the `[0, 1]` list decides that the first axis can be distributed, whereas the second axis is using one processor only and as such is aligned (non-distributed). We may now inspect the low-level `Pencil` class associated with `a`:

```
p0 = a.pencil
```

The `p0` `Pencil` object contains information about the distribution of a 2D dataarray of global shape (8, 8). The distributed array `a` has been created using the information that is in `p0`, and `p0` is used by `a` to look up information about the global array, for example:

```
>>> a.alignment
1
>>> a.global_shape
(8, 8)
>>> a.subcomm
(<mpi4py.MPI.Cartcomm at 0x10cc14a68>, <mpi4py.MPI.Cartcomm at 0x10e028690>)
>>> a.commsizes
[1, 1]
```

Naturally, the sizes of the communicators will depend on the number of processors used to run the program. If we used 4, then `a.commsizes` would return `[1, 4]`.

We note that a low-level approach to creating such a distributed array would be:

```
import numpy as np
from mpi4py_fft.pencil import Pencil, Subcomm
from mpi4py import MPI
comm = MPI.COMM_WORLD
N = (8, 8)
subcomm = Subcomm(comm, [0, 1])
p0 = Pencil(subcomm, N, axis=1)
a0 = np.zeros(p0.subshape)
```

Note that this last array `a0` would be equivalent to `a`, but it would be a pure Numpy array (created on each processor) and it would not contain any of the information about the global array that it is part of (`global_shape, pencil, subcomm, etc.`). It contains the same amount of data as `a` though and `a0` is as such a perfectly fine distributed array. Used together with `p0` it contains exactly the same information as `a`.

Since at least one axis needs to be aligned (non-distributed), a 2D array can only be distributed with one processor group. If we wanted to distribute the second axis instead of the first, then we would have done:

```
a = DistArray(N, [1, 0])
```

With the low-level approach we would have had to use `axis=0` in the creation of `p0`, as well as `[1, 0]` in the creation of `subcomm`. Another way to get the second `pencil`, that is aligned with axis 0, is to create it from `p0`:

```
p1 = p0.pencil(0)
```

Now the `p1` object will represent a (8, 8) global array distributed in the second axis.

Lets create a complete script (`pencils.py`) that fills the array `a` with the value of each processors rank (note that it would also work to follow the low-level approach and use `a0`):

```
import numpy as np
from mpi4py_fft import DistArray
from mpi4py import MPI
comm = MPI.COMM_WORLD
N = (8, 8)
a = DistArray(N, [0, 1])
a[:] = comm.Get_rank()
print(a.shape)
```

We can run it with:

```
mpirun -np 4 python pencils.py
```

and obtain the printed results from the last line (`print(a.shape)`):

```
(2, 8)
(2, 8)
(2, 8)
(2, 8)
```

The shape of the local `a` arrays is (2, 8) on all 4 processors. Now assume that we need these data aligned in the x-direction (axis=0) instead. For this to happen we need to perform a *global redistribution*. The easiest approach is then to execute the following:

```
b = a.redistribute(0)
print(b.shape)
```

which would print the following:

```
(8, 2)
(8, 2)
(8, 2)
(8, 2)
```

Under the hood the global redistribution is executed with the help of the `Transfer` class, that is designed to transfer data between any two sets of pencils, like those represented by `p0` and `p1`. With low-level API a transfer object may be created using the pencils and the datatype of the array that is to be sent:

```
transfer = p0.transfer(p1, np.float)
```

Executing the global redistribution is then simply a matter of:

```
a1 = np.zeros(p1.subshape)
transfer.forward(a, a1)
```

Now it is important to realise that the global array does not change. The local `a1` arrays will now contain the same data as `a`, only aligned differently. However, the exchange is not performed in-place. The new array is as such a copy of the original that is aligned differently. Some images, Fig. 1.1 and Fig. 1.2, can be used to illustrate:

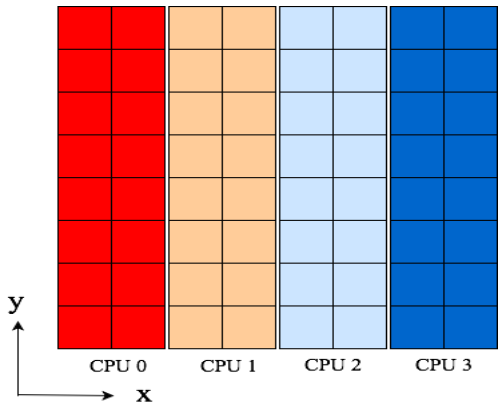


Fig. 1.1: Original 4 pencils (p0) of shape (2, 8) aligned in y-direction. Color represents rank.

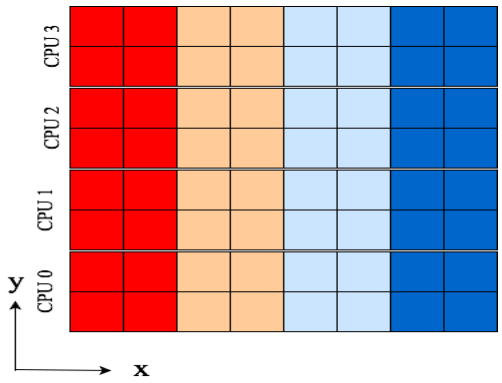


Fig. 1.2: 4 pencils (p1) of shape (8, 2) aligned in x-direction after receiving data from p0. Data is the same as in Fig. 1.1, only aligned differently.

Mathematically, we will denote the entries of a two-dimensional global array as u_{j_0,j_1}, where $j_0 \in \mathbf{j}_0 = [0, 1, \ldots, N_0 - 1]$ and $j_1 \in \mathbf{j}_1 = [0, 1, \ldots, N_1 - 1]$. The shape of the array is then (N_0, N_1). A global array u_{j_0,j_1} distributed in the first axis (as shown in Fig. 1.1) by processor group P, containing $|P|$ processors, is denoted as

$$u_{j_0/P,j_1}$$

The global redistribution, from alignment in axis 1 to alignment in axis 0, as from Fig. 1.1 to Fig. 1.2 above, is denoted as

$$u_{j_0,j_1/P} \xleftarrow[P]{1 \to 0} u_{j_0/P,j_1}$$

This operation corresponds exactly to the forward transfer defined above:

```
transfer.forward(a0, a1)
```

If we need to go the other way

$$u_{j_0/P,j_1} \xleftarrow[P]{0\rightarrow 1} u_{j_0,j_1/P}$$

this corresponds to:

```
transfer.backward(a1, a0)
```

Note that the directions (forward/backward) here depends on how the transfer object is created. Under the hood all transfers are executing calls to MPI.Alltoallw.

1.2.1 Multidimensional distributed arrays

The procedure discussed above remains the same for any type of array, of any dimensionality. With mpi4py-fft we can distribute any array of arbitrary dimensionality using any number of processor groups. We only require that the number of processor groups is at least one less than the number of dimensions, since one axis must remain aligned. Apart from this the distribution is completely configurable through the classes in the `pencil` module.

We denote a global d-dimensional array as $u_{j_0,j_1,\ldots,j_{d-1}}$, where $j_m \in \mathbf{j}_m$ for $m = [0, 1, \ldots, d-1]$. A d-dimensional array distributed with only one processor group in the first axis is denoted as $u_{j_0/P,j_1,\ldots,j_{d-1}}$. If using more than one processor group, the groups are indexed, like P_0, P_1 etc.

Lets illustrate using a 4-dimensional array with 3 processor groups. Let the array be aligned only in axis 3 first ($u_{j_0/P_0,j_1/P_1,j_2/P_2,j_3}$), and then redistribute for alignment along axes 2, 1 and finally 0. Mathematically, we will now be executing the three following global redistributions:

$$\begin{aligned} u_{j_0/P_0,j_1/P_1,j_2,j_3/P_2} &\xleftarrow[P_2]{3\rightarrow 2} u_{j_0/P_0,j_1/P_1,j_2/P_2,j_3} \\ u_{j_0/P_0,j_1,j_2/P_1,j_3/P_2} &\xleftarrow[P_1]{2\rightarrow 1} u_{j_0/P_0,j_1/P_1,j_2,j_3/P_2} \\ u_{j_0,j_1/P_0,j_2/P_1,j_3/P_2} &\xleftarrow[P_0]{1\rightarrow 0} u_{j_0/P_0,j_1,j_2/P_1,j_3/P_2} \end{aligned} \tag{1.1}$$

Note that in the first step it is only processor group P_2 that is active in the redistribution, and the output (left hand side) is now aligned in axis 2. This can be seen since there is no processor group there to share the j_2 index. In the second step processor group P_1 is the active one, and in the final step P_0.

Now, it is not necessary to use three processor groups just because we have a four-dimensional array. We could just as well have been using 2 or 1. The advantage of using more groups is that you can then use more processors in total. Assuming $N = N_0 = N_1 = N_2 = N_3$, you can use a maximum of N^p processors, where p is the number of processor groups. So for an array of shape $(8, 8, 8, 8)$ it is possible to use 8, 64 and 512 number of processors for 1, 2 and 3 processor groups, respectively. On the other hand, if you can get away with it, or if you do not have access to a great number of processors, then fewer groups are usually found to be faster for the same number of processors in total.

We can implement the global redistribution using the high-level `DistArray` class:

```
N = (8, 8, 8, 8)
a3 = DistArray(N, [0, 0, 0, 1])
a2 = a3.redistribute(2)
a1 = a2.redistribute(1)
a0 = a1.redistribute(0)
```

Note that the three redistribution steps correspond exactly to the three steps in (1.1).

Using a low-level API the same can be achieved with a little more elaborate coding. We start by creating pencils for the 4 different alignments:

```
subcomm = Subcomm(comm, [0, 0, 0, 1])
p3 = Pencil(subcomm, N, axis=3)
p2 = p3.pencil(2)
p1 = p2.pencil(1)
p0 = p1.pencil(0)
```

Here we have defined 4 different pencil groups, `p0, p1, p2, p3`, aligned in axis 0, 1, 2 and 3, respectively. Transfer objects for arrays of type `np.float` are then created as:

```
transfer32 = p3.transfer(p2, np.float)
transfer21 = p2.transfer(p1, np.float)
transfer10 = p1.transfer(p0, np.float)
```

Note that we can create transfer objects between any two pencils, not just neighbouring axes. We may now perform three different global redistributions as:

```
a0 = np.zeros(p0.subshape)
a1 = np.zeros(p1.subshape)
a2 = np.zeros(p2.subshape)
a3 = np.zeros(p3.subshape)
a0[:] = np.random.random(a0.shape)
transfer32.forward(a3, a2)
transfer21.forward(a2, a1)
transfer10.forward(a1, a0)
```

Storing this code under `pencils4d.py`, we can use 8 processors that will give us 3 processor groups with 2 processors in each group:

```
mpirun -np 8 python pencils4d.py
```

Note that with the low-level approach we can now easily go back using the reverse `backward` method of the `Transfer` objects:

```
transfer10.backward(a0, a1)
```

A different approach is also possible with the high-level API:

```
a0.redistribute(out=a1)
a1.redistribute(out=a2)
a2.redistribute(out=a3)
```

which corresponds to the backward transfers. However, with the high-level API the transfer objects are created (and deleted on exit) during the call to `redistribute` and as such this latter approach may be slightly less efficient.

1.3 Discrete Fourier Transforms

Consider first two one-dimensional arrays $\boldsymbol{u} = \{u_j\}_{j=0}^{N-1}$ and $\hat{\boldsymbol{u}} = \{\hat{u}_k\}_{k=0}^{N-1}$. We define the forward and backward Discrete Fourier transforms (DFT), respectively, as

$$\begin{aligned} \hat{u}_k &= \frac{1}{N}\sum_{j=0}^{N-1} u_j e^{-2\pi ijk/N}, \quad \forall\, k \in \mathbf{k} = 0, 1, \ldots, N-1, \\ u_j &= \sum_{k=0}^{N-1} \hat{u}_k e^{2\pi ijk/N}, \quad \forall\, j \in \mathbf{j} = 0, 1, \ldots, N-1, \end{aligned} \tag{1.2}$$

where $i = \sqrt{-1}$. Discrete Fourier transforms are computed efficiently using algorithms termed Fast Fourier Transforms, known in short as FFTs.

Note: The index set for wavenumbers $\mathbf{k}$ is usually not chosen as $[0, 1, \ldots, N-1]$, but $\mathbf{k} = [-N/2, -N/2 - 1, \ldots, N/2-1]$ for even N and $\mathbf{k} = [-(N-1)/2, -(N-1)/2+1, \ldots, (N-1)/2]$ for odd N. See numpy.fft.fftfreq. Also note that it is possible to tweak the default normalization used above when calling either forward or backward transforms.

A more compact notation is commonly used for the DFTs, where the 1D forward and backward transforms are written as

$$\begin{aligned} \hat{\boldsymbol{u}} &= \mathcal{F}(\boldsymbol{u}), \\ \boldsymbol{u} &= \mathcal{F}^{-1}(\hat{\boldsymbol{u}}). \end{aligned}$$

Numpy, Scipy, and many other scientific softwares contain implementations that make working with Fourier series simple and straight forward. These 1D Fourier transforms can be implemented easily with just Numpy as, e.g.:

```
import numpy as np
N = 16
u = np.random.random(N)
u_hat = np.fft.fft(u)
uc = np.fft.ifft(u_hat)
assert np.allclose(u, uc)
```

However, there is a minor difference. Numpy performs by default the $1/N$ scaling with the *backward* transform (`ifft`) and not the forward as shown in (1.2). These are merely different conventions and not important as long as one is aware of them. We use the scaling on the forward transform simply because this follows naturally when using the harmonic functions e^{ikx} as basis functions when solving PDEs with the spectral Galerkin method or the spectral collocation method (see chap. 3).

With mpi4py-fft the same operations take just a few more steps, because instead of executing ffts directly, like in the calls for `np.fft.fft` and `np.fft.ifft`, we need to create the objects that are to do the transforms first. We need to *plan* the transforms:

```
from mpi4py_fft import fftw
u = fftw.aligned(N, dtype=np.complex)
u_hat = fftw.aligned_like(u)
fft = fftw.fftn(u, flags=(fftw.FFTW_MEASURE,))        # plan fft
ifft = fftw.ifftn(u_hat, flags=(fftw.FFTW_ESTIMATE,)) # plan ifft
u[:] = np.random.random(N)
# Now execute the transforms
```

(continues on next page)

(continued from previous page)

```
u_hat = fft(u, u_hat, normalize=True)
uc = ifft(u_hat)
assert np.allclose(uc, u)
```

The planning of transforms makes an effort to find the fastest possible transform of the given kind. See more in *The fftw module*.

1.3.1 Multidimensional transforms

It is for multidimensional arrays that it starts to become interesting for the current software. Multidimensional arrays are a bit tedious with notation, though, especially when the number of dimensions grow. We will stick with the index notation because it is most straightforward in comparison with implementation.

We denote the entries of a two-dimensional array as u_{j_0,j_1}, which corresponds to a row-major matrix $\boldsymbol{u} = \{u_{j_0,j_1}\}_{(j_0,j_1)\in \mathbf{j}_0\times\mathbf{j}_1}$ of size $N_0 \cdot N_1$. Denoting also $\omega_m = j_m k_m/N_m$, a two-dimensional forward and backward DFT can be defined as

$$\begin{aligned}
\hat{u}_{k_0,k_1} &= \frac{1}{N_0}\sum_{j_0\in\mathbf{j}_0}\left(e^{-2\pi i\omega_0}\frac{1}{N_1}\sum_{j_1\in\mathbf{j}_1}\left(e^{-2\pi i\omega_1}u_{j_0,j_1}\right)\right), \quad \forall\,(k_0,k_1)\in\mathbf{k}_0\times\mathbf{k}_1,\\
u_{j_0,j_1} &= \sum_{k_1\in\mathbf{k}_1}\left(e^{2\pi i\omega_1}\sum_{k_0\in\mathbf{k}_0}\left(e^{2\pi i\omega_0}\hat{u}_{k_0,k_1}\right)\right), \quad \forall\,(j_0,j_1)\in\mathbf{j}_0\times\mathbf{j}_1.
\end{aligned} \tag{1.3}$$

Note that the forward transform corresponds to taking the 1D Fourier transform first along axis 1, once for each of the indices in $\mathbf{j}_0$. Afterwords the transform is executed along axis 0. The two steps are more easily understood if we break things up a little bit and write the forward transform in (1.3) in two steps as

$$\begin{aligned}
\tilde{u}_{j_0,k_1} &= \frac{1}{N_1}\sum_{j_1\in\mathbf{j}_1}u_{j_0,j_1}e^{-2\pi i\omega_1}, \quad \forall\,k_1\in\mathbf{k}_1,\\
\hat{u}_{k_0,k_1} &= \frac{1}{N_0}\sum_{j_0\in\mathbf{j}_0}\tilde{u}_{j_0,k_1}e^{-2\pi i\omega_0}, \quad \forall\,k_0\in\mathbf{k}_0.
\end{aligned} \tag{1.4}$$

The backward (inverse) transform if performed in the opposite order, axis 0 first and then 1. The order is actually arbitrary, but this is how is is usually computed. With mpi4py-fft the order of the directional transforms can easily be configured.

We can write the complete transform on compact notation as

$$\begin{aligned}
\hat{\boldsymbol{u}} &= \mathcal{F}(\boldsymbol{u}),\\
\boldsymbol{u} &= \mathcal{F}^{-1}(\hat{\boldsymbol{u}}).
\end{aligned} \tag{1.5}$$

But if we denote the two *partial* transforms along each axis as $\mathcal{F}_0$ and $\mathcal{F}_1$, we can also write it as

$$\begin{aligned}
\hat{\boldsymbol{u}} &= \mathcal{F}_0(\mathcal{F}_1(\boldsymbol{u})),\\
\boldsymbol{u} &= \mathcal{F}_1^{-1}(\mathcal{F}_0^{-1}(\hat{\boldsymbol{u}})).
\end{aligned} \tag{1.6}$$

Extension to multiple dimensions is straight forward. We denote a d-dimensional array as $u_{j_0,j_1,\ldots,j_{d-1}}$ and a partial transform of u along axis i is denoted as

$$\tilde{u}_{j_0,\ldots,k_i,\ldots,j_{d-1}} = \mathcal{F}_i(u_{j_0,\ldots,j_i,\ldots,j_{d-1}}) \tag{1.7}$$

We get the complete multidimensional transforms on short form still as (1.5), and with partial transforms as

$$\begin{aligned}
\hat{\boldsymbol{u}} &= \mathcal{F}_0(\mathcal{F}_1(\ldots\mathcal{F}_{d-1}(\boldsymbol{u})),\\
\boldsymbol{u} &= \mathcal{F}_{d-1}^{-1}(\mathcal{F}_{d-2}^{-1}(\ldots\mathcal{F}_0^{-1}(\hat{\boldsymbol{u}}))).
\end{aligned} \tag{1.8}$$

Multidimensional transforms are straightforward to implement in Numpy

```
import numpy as np
M, N = 16, 16
u = np.random.random((M, N))
u_hat = np.fft.rfftn(u)
uc = np.fft.irfftn(u_hat)
assert np.allclose(u, uc)
```

1.3.2 The `fftw` module

The `fftw` module provides an interface to most of the FFTW library. In the `fftw.xfftn` submodule there are planner functions for:

- `fftn()` - complex-to-complex forward Fast Fourier Transforms
- `ifftn()` - complex-to-complex backward Fast Fourier Transforms
- `rfftn()` - real-to-complex forward FFT
- `irfftn()` - complex-to-real backward FFT
- `dctn()` - real-to-real Discrete Cosine Transform (DCT)
- `idctn()` - real-to-real inverse DCT
- `dstn()` - real-to-real Discrete Sine Transform (DST)
- `idstn()` - real-to-real inverse DST
- `hfftn()` - complex-to-real forward FFT with Hermitian symmetry
- `ihfftn()` - real-to-complex backward FFT with Hermitian symmetry

All these transform functions return instances of one of the classes `fftwf_xfftn.FFT`, `fftw_xfftn.FFT` or `fftwl_xfftn.FFT`, depending on the requested precision being single, double or long double, respectively. Except from precision, the tree classes are identical. All transforms are non-normalized by default. Note that all these functions are *planners*. They do not execute the transforms, they simply return an instance of a class that can do it (see docstrings of each function for usage). For quick reference, the 2D transform *shown for Numpy* can be done using `fftw` as:

```
from mpi4py_fft.fftw import rfftn as plan_rfftn, irfftn as plan_irfftn
from mpi4py_fft.fftw import FFTW_ESTIMATE
rfftn = plan_rfftn(u.copy(), flags=(FFTW_ESTIMATE,))
irfftn = plan_irfftn(u_hat.copy(), flags=(FFTW_ESTIMATE,))
u_hat = rfftn(uc, normalize=True)
uu = irfftn(u_hat)
assert np.allclose(uu, uc)
```

Note that since all the functions in the above list are planners, an extra step is required in comparison with Numpy. Also note that we are using copies of the `u` and `u_hat` arrays in creating the plans. This is done because the provided arrays will be used under the hood as work arrays for the `rfftn()` and `irfftn()` functions, and the work arrays may be destroyed upon creation.

The real-to-real transforms are by FFTW defined as one of (see definitions and extended definitions)

- FFTW_REDFT00
- FFTW_REDFT01
- FFTW_REDFT10
- FFTW_REDFT11

- FFTW_RODFT00
- FFTW_RODFT01
- FFTW_RODFT10
- FFTW_RODFT11

Different real-to-real cosine and sine transforms may be combined into one object using `factory.get_planned_FFT()` with a list of different transform kinds. However, it is not possible to combine, in one single object, real-to-real transforms with real-to-complex. For such transforms more than one object is required.

1.4 Parallel Fast Fourier Transforms

Parallel FFTs are computed through a combination of *global redistributions* and *serial transforms*. In mpi4py-fft the interface to performing such parallel transforms is the `mpifft.PFFT` class. The class is highly configurable and best explained through a few examples.

1.4.1 Slab decomposition

With slab decompositions we use only one group of processors and distribute only one index of a multidimensional array at the time.

Consider the complete transform of a three-dimensional array of random numbers, and of shape (128, 128, 128). We can plan the transform of such an array with the following code snippet:

```
import numpy as np
from mpi4py import MPI
from mpi4py_fft import PFFT, newDistArray
N = np.array([128, 128, 128], dtype=int)
fft = PFFT(MPI.COMM_WORLD, N, axes=(0, 1, 2), dtype=np.float, grid=(-1,))
```

Here the signature `N, axes=(0, 1, 2), dtype=np.float, grid=(-1,)` tells us that the created `fft` instance is *planned* such as to slab distribute (along first axis) and transform any 3D array of shape `N` and type `np.float`. Furthermore, we plan to transform axis 2 first, and then 1 and 0, which is exactly the reverse order of `axes=(0, 1, 2)`. Mathematically, the planned transform corresponds to

$$\begin{aligned}
\tilde{u}_{j_0/P,k_1,k_2} &= \mathcal{F}_1(\mathcal{F}_2(u_{j_0/P,j_1,j_2})),\\
\tilde{u}_{j_0,k_1/P,k_2} &\xleftarrow{\frac{1\rightarrow 0}{P}} \tilde{u}_{j_0/P,k_1,k_2},\\
\hat{u}_{k_0,k_1/P,k_2} &= \mathcal{F}_0(\tilde{u}_{j_0,k_1/P,k_2}).
\end{aligned}$$

Note that axis 0 is distributed on the input array and axis 1 on the output array. In the first step above we compute the transforms along axes 2 and 1 (in that order), but we cannot compute the serial transform along axis 0 since the global array is distributed in that direction. We need to perform a global redistribution, the middle step, that realigns the global data such that it is aligned in axes 0. With data aligned in axis 0, we can perform the final transform $\mathcal{F}_0$ and be done with it.

Assume now that all the code in this section is stored to a file named `pfft_example.py`, and add to the above code:

```
u = newDistArray(fft, False)
u[:] = np.random.random(u.shape).astype(u.dtype)
u_hat = fft.forward(u, normalize=True) # Note that normalize=True is default and can be␣
↪omitted
```

(continues on next page)

(continued from previous page)

```
uj = np.zeros_like(u)
uj = fft.backward(u_hat, uj)
assert np.allclose(uj, u)
print(MPI.COMM_WORLD.Get_rank(), u.shape)
```

Running this code with two processors (`mpirun -np 2 python pfft_example.py`) should raise no exception, and the output should be:

```
1 (64, 128, 128)
0 (64, 128, 128)
```

This shows that the first index has been shared between the two processors equally. The array `u` thus corresponds to $u_{j_0/P,j_1,j_2}$. Note that the `newDistArray()` function returns a `DistArray` object, which in turn is a subclassed Numpy ndarray. The `newDistArray()` function uses `fft` to determine the size and type of the created distributed array, i.e., (64, 128, 128) and `np.float` for both processors. The `False` argument indicates that the shape and type should be that of the input array, as opposed to the output array type ($\hat{u}_{k_0,k_1/P,k_2}$ that one gets with `True`).

Note that because the input array is of real type, and not complex, the output array will be of global shape:

```
128, 128, 65
```

The output array will be distributed in axis 1, so the output array shape should be (128, 64, 65) on each processor. We check this by adding the following code and rerunning:

```
u_hat = newDistArray(fft, True)
print(MPI.COMM_WORLD.Get_rank(), u_hat.shape)
```

leading to an additional print of:

```
1 (128, 64, 65)
0 (128, 64, 65)
```

To distribute in the first axis first is default and most efficient for row-major C arrays. However, we can easily configure the `fft` instance by modifying the axes keyword. Changing for example to:

```
fft = PFFT(MPI.COMM_WORLD, N, axes=(2, 0, 1), dtype=np.float)
```

and axis 1 will be transformed first, such that the global output array will be of shape (128, 65, 128). The distributed input and output arrays will now have shape:

```
0 (64, 128, 128)
1 (64, 128, 128)

0 (128, 33, 128)
1 (128, 32, 128)
```

Note that the input array will still be distributed in axis 0 and the output in axis 1. This order of distribution can be tweaked using the *grid* keyword. Setting *grid=(1, 1, -1)* will force the last axis to be distributed on the input array.

Another way to tweak the distribution is to use the `Subcomm` class directly:

```
from mpi4py_fft.pencil import Subcomm
subcomms = Subcomm(MPI.COMM_WORLD, [1, 0, 1])
fft = PFFT(subcomms, N, axes=(0, 1, 2), dtype=np.float)
```

Here the `subcomms` tuple will decide that axis 1 should be distributed, because the only zero in the list `[1, 0, 1]` is along axis 1. The ones determine that axes 0 and 2 should use one processor each, i.e., they should be non-distributed.

The `PFFT` class has a few additional keyword arguments that one should be aware of. The default behaviour of `PFFT` is to use one transform object for each axis, and then use these sequentially. Setting `collapse=True` will attempt to minimize the number of transform objects by combining whenever possible. Take our example, the array $u_{j_0/P,j_1,j_2}$ can transform along both axes 1 and 2 simultaneously, without any intermediate global redistributions. By setting `collapse=True` only one object of `rfftn(u, axes=(1, 2))` will be used instead of two (like `fftn(rfftn(u, axes=2), axes=1)`). Note that a collapse can also be configured through the `axes` keyword, using:

```
fft = PFFT(MPI.COMM_WORLD, N, axes=((0,), (1, 2)), dtype=np.float)
```

will collapse axes 1 and 2, just like one would obtain with `collapse=True`.

If serial transforms other than `fftn()`/`rfftn()` and `ifftn()`/`irfftn()` are required, then this can be achieved using the `transforms` keyword and a dictionary pointing from axes to the type of transform. We can for example combine real-to-real with real-to-complex transforms like this:

```
from mpi4py_fft.fftw import rfftn, irfftn, dctn, idctn
import functools
dct = functools.partial(dctn, type=3)
idct = functools.partial(idctn, type=3)
transforms = {(0,): (rfftn, irfftn), (1, 2): (dct, idct)}
r2c = PFFT(MPI.COMM_WORLD, N, axes=((0,), (1, 2)), transforms=transforms)
u = newDistArray(r2c, False)
u[:] = np.random.random(u.shape).astype(u.dtype)
u_hat = r2c.forward(u)
uj = np.zeros_like(u)
uj = r2c.backward(u_hat, uj)
assert np.allclose(uj, u)
```

As a more complex example consider a 5-dimensional array where for some reason you need to perform discrete cosine transforms in axes 1 and 2, discrete sine transforms in axes 3 and 4, and a regular Fourier transform in the first axis. Here it makes sense to collapse the (1, 2) and (3, 4) axes, which leaves only the first axis uncollapsed. Hence we can then only use one processor group and a slab decomposition, whereas without collapsing we could have used four groups. A parallel transform object can be created and tested as:

```
N = (5, 6, 7, 8, 9)
dctn = functools.partial(fftw.dctn, type=3)
idctn = functools.partial(fftw.idctn, type=3)
dstn = functools.partial(fftw.dstn, type=3)
idstn = functools.partial(fftw.idstn, type=3)
fft = PFFT(MPI.COMM_WORLD, N, ((0,), (1, 2), (3, 4)), grid=(-1,),
           transforms={(1, 2): (dctn, idctn), (3, 4): (dstn, idstn)})

A = newDistArray(fft, False)
A[:] = np.random.random(A.shape)
C = fftw.aligned_like(A)
B = fft.forward(A)
C = fft.backward(B, C)
assert np.allclose(A, C)
```

1.4.2 Pencil decomposition

A pencil decomposition uses two groups of processors. Each group then is responsible for distributing one index set each of a multidimensional array. We can perform a pencil decomposition simply by running the first example from the previous section, but now with 4 processors. To remind you, we put this in `pfft_example.py`, where now `grid=(-1,)` has been removed in the PFFT calling:

```
import numpy as np
from mpi4py import MPI
from mpi4py_fft import PFFT, newDistArray

N = np.array([128, 128, 128], dtype=int)
fft = PFFT(MPI.COMM_WORLD, N, axes=(0, 1, 2), dtype=np.float)
u = newDistArray(fft, False)
u[:] = np.random.random(u.shape).astype(u.dtype)
u_hat = fft.forward(u)
uj = np.zeros_like(u)
uj = fft.backward(u_hat, uj)
assert np.allclose(uj, u)
print(MPI.COMM_WORLD.Get_rank(), u.shape)
```

The output of running `mpirun -np 4 python pfft_example.py` will then be:

```
0 (64, 64, 128)
2 (64, 64, 128)
3 (64, 64, 128)
1 (64, 64, 128)
```

Note that now both the two first index sets are shared, so we have a pencil decomposition. The shared input array is now denoted as $u_{j_0/P_0,j_1/P_1,j2}$ and the complete forward transform performs the following 5 steps:

$$
\begin{aligned}
\tilde{u}_{j_0/P_0,j_1/P_1,k_2} &= \mathcal{F}_2(u_{j_0/P_0,j_1/P_1,j_2}),\\
\tilde{u}_{j_0/P_0,j_1,k_2/P_1} &\xleftarrow[P_1]{2\to 1} \tilde{u}_{j_0/P_0,j_1/P_1,k_2},\\
\tilde{u}_{j_0/P_0,k_1,k_2/P_1} &= \mathcal{F}_1(\tilde{u}_{j_0/P_0,j_1,k_2/P_1}),\\
\tilde{u}_{j_0,k_1/P_0,k_2/P_1} &\xleftarrow[P_0]{1\to 0} \tilde{u}_{j_0/P_0,k_1,k_2/P_1},\\
\hat{u}_{k_0,k_1/P_0,k_2/P_1} &= \mathcal{F}_0(\tilde{u}_{j_0,k_1/P_0,k_2/P_1}).
\end{aligned}
$$

Like for the slab decomposition, the order of the different steps is configurable. Simply change the value of `axes`, e.g., as:

```
fft = PFFT(MPI.COMM_WORLD, N, axes=(2, 0, 1), dtype=np.float)
```

and the input and output arrays will be of shape:

```
3 (64, 128, 64)
2 (64, 128, 64)
1 (64, 128, 64)
0 (64, 128, 64)

3 (64, 32, 128)
2 (64, 32, 128)
1 (64, 33, 128)
0 (64, 33, 128)
```

We see that the input array is aligned in axis 1, because this is the direction transformed first.

1.4.3 Convolution

Working with Fourier one sometimes need to transform the product of two or more functions, like

$$\widehat{ab}_k = \int_0^{2\pi} ab e^{-ikx} dx, \quad \forall k \in [-N/2, \ldots, N/2-1] \tag{1.9}$$

computed with DFT as

$$\widehat{ab}_k = \frac{1}{N} \sum_{j=0}^{N-1} a_j b_j e^{-2\pi ijk/N}, \quad \forall\, k \in [-N/2, \ldots, N/2-1]. \tag{1.10}$$

Note: We are here assuming an even number N and use wavenumbers centered around zero.

If a and b are two Fourier series with their own coefficients:

$$\begin{aligned} a &= \sum_{p=-N/2}^{N/2-1} \hat{a}_p e^{ipx}, \\ b &= \sum_{q=-N/2}^{N/2-1} \hat{b}_q e^{iqx}, \end{aligned} \tag{1.11}$$

then we can insert for the two sums from (1.11) in (1.9) and get

$$\begin{aligned} \widehat{ab}_k &= \int_0^{2\pi} \left(\sum_{p=-N/2}^{N/2-1} \hat{a}_p e^{ipx} \sum_{q=-N/2}^{N/2-1} \hat{b}_q e^{iqx} \right) e^{-ikx} dx, \quad \forall\, k \in [-N/2, \ldots, N/2-1] \\ \widehat{ab}_k &= \sum_{p=-N/2}^{N/2-1} \sum_{q=-N/2}^{N/2-1} \hat{a}_p \hat{b}_q \int_0^{2\pi} e^{-i(p+q-k)x} dx, \quad \forall\, k \in [-N/2, \ldots, N/2-1] \end{aligned} \tag{1.12}$$

The final integral is 2π for $p+q=k$ and zero otherwise. Consequently, we get

$$\widehat{ab}_k = 2\pi \sum_{p=-N/2}^{N/2-1} \sum_{q=-N/2}^{N/2-1} \hat{a}_p \hat{b}_q \delta_{p+q,k}, \quad \forall\, k \in [-N/2, \ldots, N/2-1] \tag{1.13}$$

Unfortunately, the convolution sum (1.13) is very expensive to compute, and the direct application of (1.10) leads to aliasing errors. Luckily there is a fast approach that eliminates aliasing as well.

The fast, alias-free, approach makes use of the FFT and zero-padded coefficient vectors. The idea is to zero-pad $\hat{a}$ and $\hat{b}$ in spectral space such that we get the extended sums

$$\begin{aligned} A_j &= \sum_{p=-M/2}^{M/2-1} \hat{\hat{a}}_p e^{2\pi ipj/M}, \\ B_j &= \sum_{q=-M/2}^{M/2-1} \hat{\hat{b}}_q e^{2\pi iqj/M}, \end{aligned}$$

where $M > N$ and where the coefficients have been zero-padded such that

$$\hat{\hat{a}}_p = \begin{cases} \hat{a}_p, & \forall |p| \leq N/2 \\ 0, & \forall |p| N/2 \end{cases}$$

Now compute the nonlinear term in the larger physical space and compute the convolution as

$$\widehat{ab}_k = \frac{1}{M}\sum_{j=0}^{M-1} A_j B_j e^{-2\pi ikj/M}, \quad \forall\, k \in [-M/2, \ldots, M/2-1] \tag{1.14}$$

Finally, truncate the vector $\widehat{ab}_k$ to the original range $k \in [-N/2, \ldots, N/2-1]$, simply by eliminating all the wavenumbers higher than $|N/2|$.

With mpi4py-fft we can compute this convolution using the `padding` keyword of the `PFFT` class:

```
import numpy as np
from mpi4py_fft import PFFT, newDistArray
from mpi4py import MPI

comm = MPI.COMM_WORLD
N = (128, 128)   # Global shape in physical space
fft = PFFT(comm, N, padding=[1.5, 1.5], dtype=np.complex)

# Create arrays in normal spectral space
a_hat = newDistArray(fft, True)
b_hat = newDistArray(fft, True)
a_hat[:] = np.random.random(a_hat.shape) + np.random.random(a_hat.shape)*1j
b_hat[:] = np.random.random(a_hat.shape) + np.random.random(a_hat.shape)*1j

# Transform to real space with padding
a = newDistArray(fft, False)
b = newDistArray(fft, False)
assert a.shape == (192//comm.Get_size(), 192)
a = fft.backward(a_hat, a)
b = fft.backward(b_hat, b)

# Do forward transform with truncation
ab_hat = fft.forward(a*b)
```

Note: The padded instance of the `PFFT` class is often used in addition to a regular non-padded class. The padded version is then used to handle non-linearities, whereas the non-padded takes care of the rest, see demo.

1.5 Storing datafiles

mpi4py-fft works with regular Numpy arrays. However, since arrays in parallel can become very large, and the arrays live on multiple processors, we require parallel IO capabilities that goes beyond Numpys regular methods. In the `mpi4py_fft.io` module there are two helper classes for dumping dataarrays to either HDF5 or NetCDF format:

- `HDF5File`
- `NCFile`

Both classes have one `write` and one `read` method that stores or reads data in parallel. A simple example of usage is:

```
from mpi4py import MPI
import numpy as np
from mpi4py_fft import PFFT, HDF5File, NCFile, newDistArray
```

(continues on next page)

(continued from previous page)

```
N = (128, 256, 512)
T = PFFT(MPI.COMM_WORLD, N)
u = newDistArray(T, forward_output=False)
v = newDistArray(T, forward_output=False, val=2)
u[:] = np.random.random(u.shape)
# Store by first creating output files
fields = {'u': [u], 'v': [v]}
f0 = HDF5File('h5test.h5', mode='w')
f1 = NCFile('nctest.nc', mode='w')
f0.write(0, fields)
f1.write(0, fields)
v[:] = 3
f0.write(1, fields)
f1.write(1, fields)
```

Note that we are here creating two datafiles `h5test.h5` and `nctest.nc`, for storing in HDF5 or NetCDF4 formats respectively. Normally, one would be satisfied using only one format, so this is only for illustration. We store the fields `u` and `v` on three different occasions, so the datafiles will contain three snapshots of each field `u` and `v`.

Also note that an alternative and perhaps simpler approach is to just use the `write` method of each distributed array:

```
u.write('h5test.h5', 'u', step=2)
v.write('h5test.h5', 'v', step=2)
u.write('nctest.nc', 'u', step=2)
v.write('nctest.nc', 'v', step=2)
```

The two different approaches can be used on the same output files.

The stored dataarrays can also be retrieved later on:

```
u0 = newDistArray(T, forward_output=False)
u1 = newDistArray(T, forward_output=False)
u0.read('h5test.h5', 'u', 0)
u1.read('h5test.h5', 'u', 1)
# or alternatively for netcdf
#u0.read('nctest.nc', 'u', 0)
#u1.read('nctest.nc', 'u', 1)
```

Note that one does not have to use the same number of processors when retrieving the data as when they were stored.

It is also possible to store only parts of the, potentially large, arrays. Any chosen slice may be stored, using a *global* view of the arrays. It is possible to store both complete fields and slices in one single call by using the following appraoch:

```
f2 = HDF5File('variousfields.h5', mode='w')
fields = {'u': [u,
                (u, [slice(None), slice(None), 4]),
                (u, [5, 5, slice(None)])],
          'v': [v,
                (v, [slice(None), 6, slice(None)])]}
f2.write(0, fields)
f2.write(1, fields)
```

Alternatively, one can use the write method of each field with the `global_slice` keyword argument:

```
u.write('variousfields.h5', 'u', 2)
u.write('variousfields.h5', 'u', 2, global_slice=[slice(None), slice(None), 4])
u.write('variousfields.h5', 'u', 2, global_slice=[5, 5, slice(None)])
v.write('variousfields.h5', 'v', 2)
v.write('variousfields.h5', 'v', 2, global_slice=[slice(None), 6, slice(None)])
```

In the end this will lead to an hdf5-file with groups:

```
variousfields.h5/
├─ u/
|  ├─ 1D/
|  |  └─ 5_5_slice/
|  |     ├─ 0
|  |     ├─ 1
|  |     └─ 3
|  ├─ 2D/
|  |  └─ slice_slice_4/
|  |     ├─ 0
|  |     ├─ 1
|  |     └─ 2
|  ├─ 3D/
|  |  ├─ 0
|  |  ├─ 1
|  |  └─ 2
|  └─ mesh/
|     ├─ x0
|     ├─ x1
|     └─ x2
└─ v/
   ├─ 2D/
   |  └─ slice_6_slice/
   |     ├─ 0
   |     ├─ 1
   |     └─ 2
   ├─ 3D/
   |  ├─ 0
   |  ├─ 1
   |  └─ 2
   └─ mesh/
      ├─ x0
      ├─ x1
      └─ x2
```

Note that a mesh is stored along with each group of data. This mesh can be given in two different ways when creating the datafiles:

1) A sequence of 2-tuples, where each 2-tuple contains the (origin, length) of the domain along its dimension. For example, a uniform mesh that originates from the origin, with lengths $\pi, 2\pi, 3\pi$, can be given when creating the output file as:

```
f0 = HDF5File('filename.h5', domain=((0, pi), (0, 2*np.pi), (0, 3*np.pi)))

or, using the write method of the distributed array:
```

(continues on next page)

(continued from previous page)

```
u.write('filename.h5', 'u', 0, domain=((0, pi), (0, 2*np.pi), (0, 3*np.pi)))
```

2) A sequence of arrays giving the coordinates for each dimension. For example:

```
d = (np.arange(N[0], dtype=np.float)*1*np.pi/N[0],
     np.arange(N[1], dtype=np.float)*2*np.pi/N[1],
     np.arange(N[2], dtype=np.float)*2*np.pi/N[2])
f0 = HDF5File('filename.h5', domain=d)
```

With NetCDF4 the layout is somewhat different. For `variousfields` above, if we were using `NCFile` instead of `HDF5File`, we would get a datafile that with `ncdump -h variousfields.nc` would look like:

```
netcdf variousfields {
dimensions:
        time = UNLIMITED ; // (3 currently)
        x = 128 ;
        y = 256 ;
        z = 512 ;
variables:
        double time(time) ;
        double x(x) ;
        double y(y) ;
        double z(z) ;
        double u(time, x, y, z) ;
        double u_slice_slice_4(time, x, y) ;
        double u_5_5_slice(time, z) ;
        double v(time, x, y, z) ;
        double v_slice_6_slice(time, x, z) ;
}
```

1.5.1 Postprocessing

Dataarrays stored to HDF5 files can be visualized using both Paraview and Visit, whereas NetCDF4 files can at the time of writing only be opened with Visit.

To view the HDF5-files we first need to generate some light-weight *xdmf*-files that can be understood by both Paraview and Visit. To generate such files, simply throw the module `io.generate_xdmf` on the HDF5-files:

```
from mpi4py_fft.io import generate_xdmf
generate_xdmf('variousfields.h5')
```

This will create a number of xdmf-files, one for each group that contains 2D or 3D data:

```
variousfields.xdmf
variousfields_slice_slice_4.xdmf
variousfields_slice_6_slice.xdmf
```

These files can be opened directly in Paraview. However, note that for Visit, one has to generate the files using:

```
generate_xdmf('variousfields.h5', order='visit')
```

because for some reason Paraview and Visit require the mesh in the xdmf-files to be stored in opposite order.

1.6 Installation

Mpi4py-fft has a few dependencies

- mpi4py
- FFTW (serial)
- numpy
- cython (build dependency)
- h5py (runtime dependency, optional)
- netCDF4 (runtime dependency, optional)

that are mostly straight-forward to install, or already installed in most Python environments. The first two are usually most troublesome. Basically, for mpi4py you need to have a working MPI installation, whereas FFTW is available on most high performance computer systems. If you are using conda, then all you need to install a fully functional mpi4py-fft, with all the above dependencies, is

```
conda install -c conda-forge mpifpy-fft h5py=*=mpi*
```

You probably want to install into a fresh environment, though, which can be achieved with

```
conda create --name mpi4py-fft -c conda-forge mpi4py-fft
conda activate mpi4py-fft
```

Note that this gives you mpi4py-fft with default settings. This means that you will probably get the openmpi backend. To make a specific choice of backend just specify which, like this

```
conda create --name mpi4py-fft -c conda-forge mpi4py-fft mpich
```

If you do not use conda, then you need to make sure that MPI and FFTW are installed by some other means. You can then install any version of mpi4py-fft hosted on pypi using pip

```
pip install mpi4py-fft
```

whereas the following will install the latest version from github

```
pip install git+https://github.com/mpi4py/mpi4py-fft@master
```

You can also build mpi4py-fft yourselves from the top directory, after cloning or forking

```
pip install .
```

or using conda-build with the recipes in folder `conf/`

```
conda build -c conda-forge conf/
conda create --name mpi4py-fft -c conda-forge mpi4py-fft --use-local
conda activate mpi4py-fft
```

1.6.1 Additional dependencies

For storing and retrieving data you need either HDF5 or netCDF4, compiled with support for MPI. Both are available with parallel support on conda-forge and can be installed into the current conda environment as

```
conda install -c conda-forge h5py=*=mpi* netcdf4=*=mpi*
```

Note that parallel HDF5 and NetCDF4 often are available as optimized modules on supercomputers. Otherwise, see the respective packages for how to install with support for MPI.

1.6.2 Test installation

After installing (from source) it may be a good idea to run all the tests located in the `tests` folder. A range of tests may be run using the `runtests.sh` script

```
conda install scipy, coverage
cd tests/
./runtests.sh
```

This test-suit is run automatically on every commit to github, see, e.g.,

1.7 How to cite?

Please cite mpi4py-fft using

```
@article{jpdc_fft,
    author = {{Dalcin, Lisandro and Mortensen, Mikael and Keyes, David E}},
    year = {{2019}},
    title = {{Fast parallel multidimensional FFT using advanced MPI}},
    journal = {{Journal of Parallel and Distributed Computing}},
    doi = {10.1016/j.jpdc.2019.02.006}
}
@electronic{mpi4py-fft,
    author = {{Lisandro Dalcin and Mikael Mortensen}},
    title = {{mpi4py-fft}},
    url = {{https://github.com/mpi4py/mpi4py-fft}}
}
```

1.8 How to contribute?

Mpi4py-fft is an open source project and anyone is welcome to contribute. An easy way to get started is by suggesting a new enhancement on the issue tracker. If you have found a bug, then either report this on the issue tracker, er even better, make a fork of the repository, fix the bug and then create a pull request to get the fix into the master branch.

1.9 Indices and tables

- genindex
- modindex
- search