PencilArrays.jl: highly-scalable MPI-distributed arrays in Julia

Juan Ignacio Polanco
Laboratoire de Mécanique des Fluides et d'Acoustique (LMFA)
École Centrale de Lyon

Journée Julia
Paris, 10 juin 2022

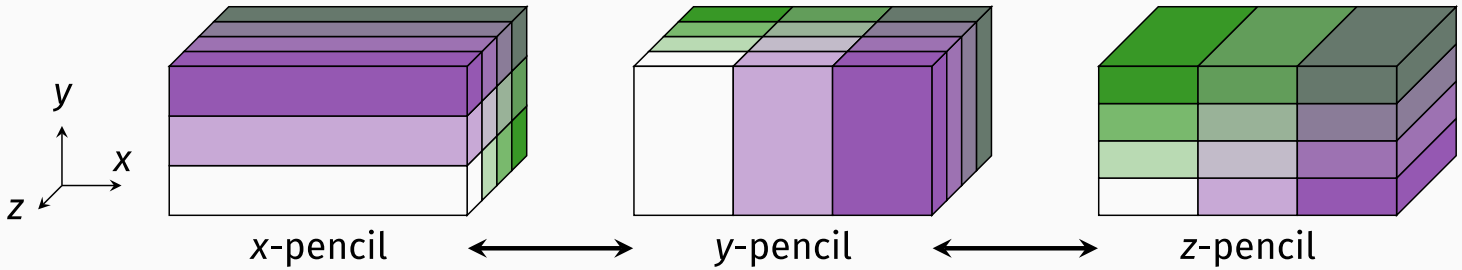
What is PencilArrays.jl / PencilFFTs.jl?

A set of open-source Julia libraries for **conveniently** and **efficiently** dealing with **distributed data**.

Aimed at numerically solving PDEs on **massively-parallel computers**.

Well-adapted for describing **structured grids**.

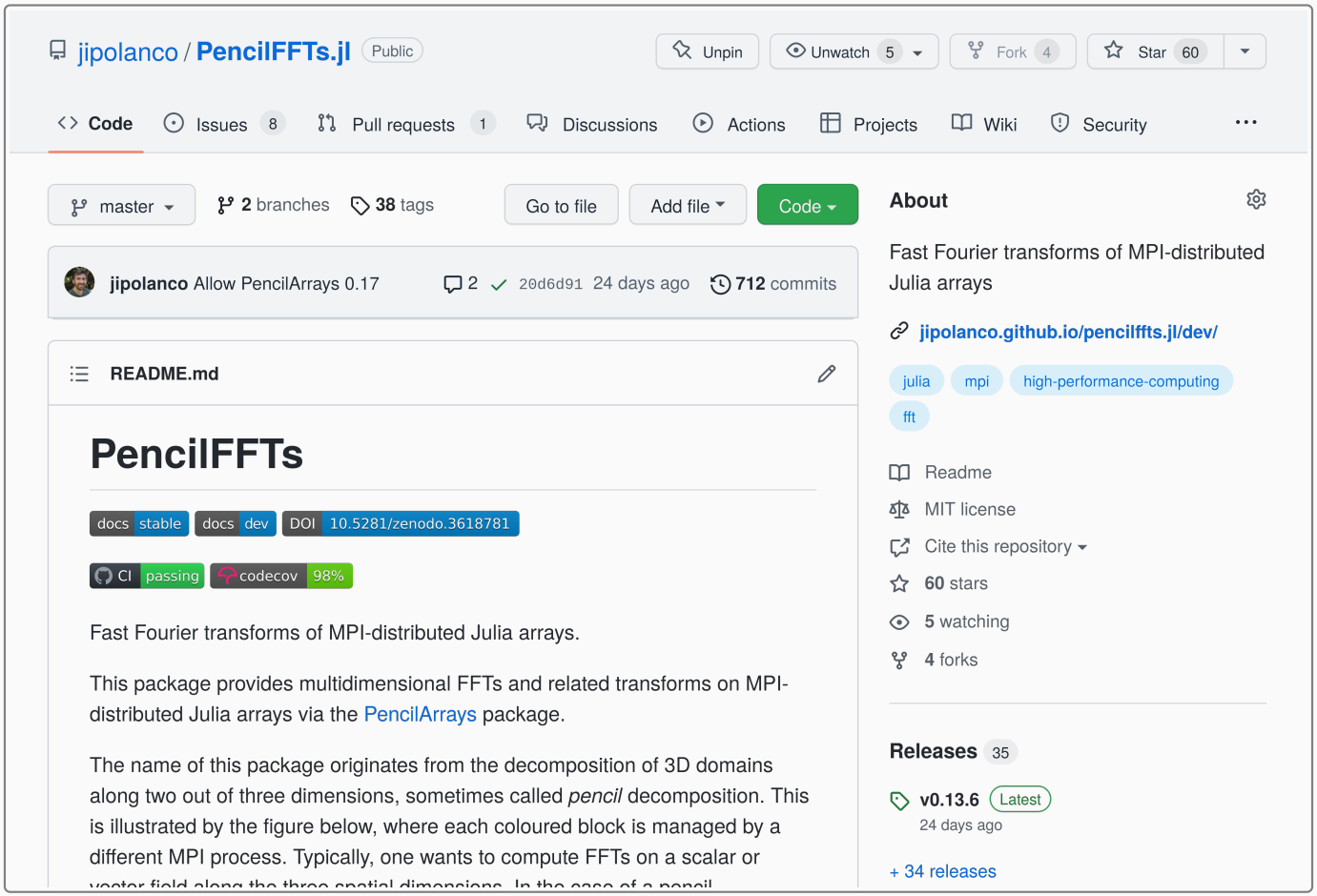
Developed with **spectral methods** in mind, but also useful for other approaches.



Sources and documentation…

https://github.com/jipolanco/PencilArrays.jl

https://github.com/jipolanco/PencilFFTs.jl



(Rough) outline of this talk

1. Motivation
2. Main features of PencilArrays.jl and PencilFFTs.jl
3. Some examples
4. Summary and perspectives

Motivation

My initial motivations

Perform **massively-parallel** numerical simulations of PDEs
(e.g. incompressible Navier–Stokes equations)

Requirements

- **Simple geometries**
  → structured grids
  → pseudo-spectral methods (FFTs / global operations)
- **Large resolutions & parallelisation**
  → distribute geometry over **several dimensions** at a time
- **Same performance** as classical compiled languages

Context: experience with MPI-parallel Navier–Stokes solvers in Fortran and C++.

Before going parallel...

what does the **(Fourier) pseudo-spectral method** look like?

1D Burgers equation for $u(x,t)$

$$\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial u^2}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \quad \text{with } 2\pi\text{-periodicity in } x$$

Write the solution as a (truncated) Fourier series:

$$u(x,t) = \sum_{k=-N}^{N} \hat{u}_k(t)\, e^{\mathrm{i}kx} \quad \rightarrow \quad \partial_t \hat{u}_k + \frac{\mathrm{i}k}{2} \mathcal{F}\left[u^2\right]_k = -\nu k^2 \hat{u}_k$$

1D Burgers equation for $u(x,t)$

$$\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial u^2}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \quad \text{with } 2\pi\text{-periodicity in } x$$

Write the solution as a (truncated) Fourier series:

$$u(x,t) = \sum_{k=-N}^{N} \hat{u}_k(t)\, e^{\mathrm{i}kx} \quad \rightarrow \quad \partial_t \hat{u}_k + \frac{\mathrm{i}k}{2} \mathcal{F}\left[u^2\right]_k = -\nu k^2 \hat{u}_k$$

The non-linear term $\mathcal{F}\left[u^2\right]_k$ is best computed using FFTs:

1. Transform to **physical space**: $\hat{u}_k \xrightarrow{\text{inverse FFT}} u_i = u(x_i)$
2. Compute u_i^2 at each grid point x_i
3. Transform to **Fourier space**: $u_i^2 \xrightarrow{\text{direct FFT}} \mathcal{F}\left[u^2\right]_k$

1D Burgers equation for $u(x,t)$

$$\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial u^2}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \quad \text{with } 2\pi\text{-periodicity in } x$$

Write the solution as a (truncated) Fourier series:

$$u(x,t) = \sum_{k=-N}^{N} \hat{u}_k(t)\, e^{\mathrm{i}kx} \quad \rightarrow \quad \partial_t \hat{u}_k + \frac{\mathrm{i}k}{2} \mathcal{F}\left[u^2\right]_k = -\nu k^2 \hat{u}_k$$

The non-linear term $\mathcal{F}\left[u^2\right]_k$ is best computed using FFTs:

1. Transform to **physical space**: $\hat{u}_k \xrightarrow{\text{inverse FFT}} u_i = u(x_i)$
2. Compute u_i^2 at each grid point x_i
3. Transform to **Fourier space**: $u_i^2 \xrightarrow{\text{direct FFT}} \mathcal{F}\left[u^2\right]_k$

Note that FFTs are **global** operations!

The idea is the same in more dimensions (say $N = 3$)

Consider a **3D-periodic** scalar field $u(x, y, z) \leftrightarrow \hat{u}(k_x, k_y, k_z)$.

To go from **physical** to **Fourier** space, I need to:

1. Transform along x:

$$u(x, y, z) \xrightarrow{\text{FFT}(x)} \hat{u}(k_x, y, z) \text{ for all } (y, z)$$

2. Transform along y:

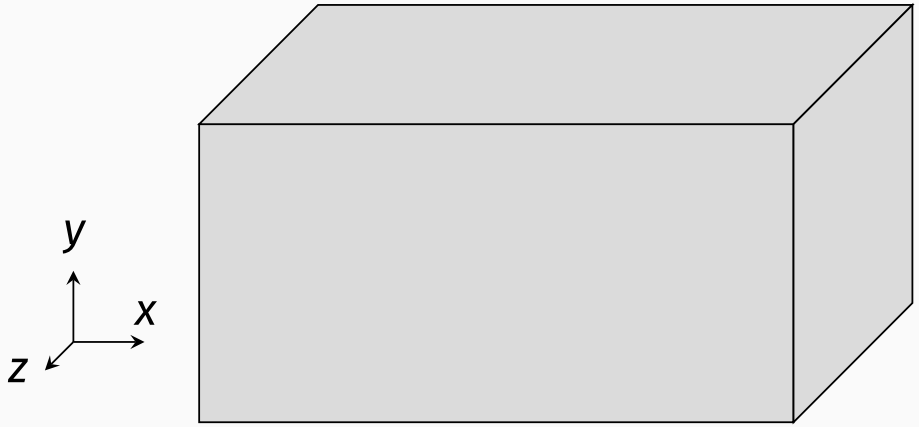
$$\hat{u}(k_x, y, z) \xrightarrow{\text{FFT}(y)} \hat{u}(k_x, k_y, z) \text{ for all } (k_x, z)$$

3. Transform along z:

$$\hat{u}(k_x, k_y, z) \xrightarrow{\text{FFT}(z)} \hat{u}(k_x, k_y, k_z) \text{ for all } (k_x, k_y)$$

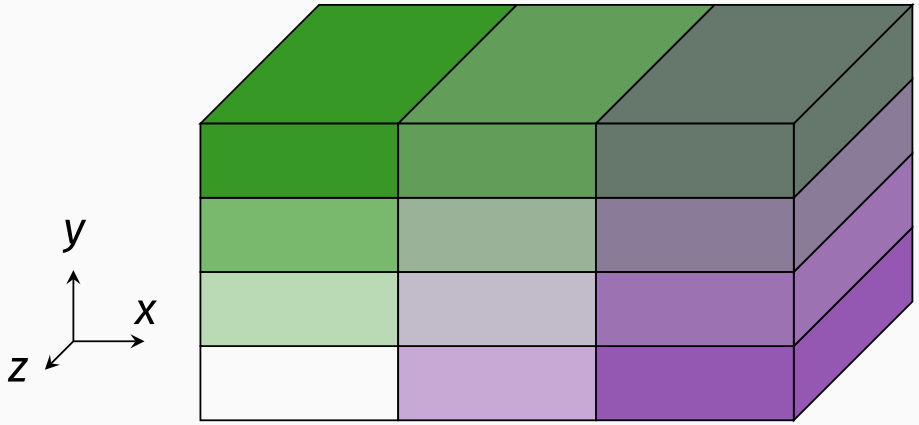
What does this mean for (MPI-style) parallelisation?

Parallelisation based on **domain decomposition**:



What does this mean for (MPI-style) parallelisation?

Parallelisation based on **domain decomposition**:



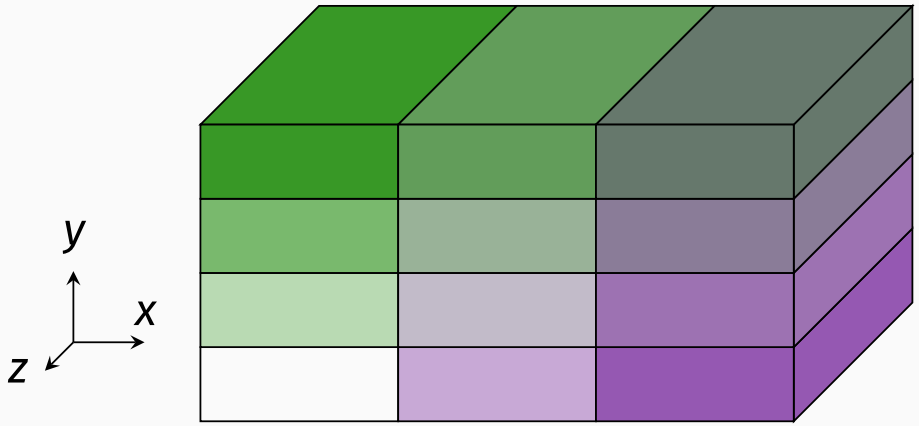
1 brick = 1 MPI process

decomposition along (x, y)

→ 2D (“pencil”) decomposition

What does this mean for (MPI-style) parallelisation?

Parallelisation based on **domain decomposition**:



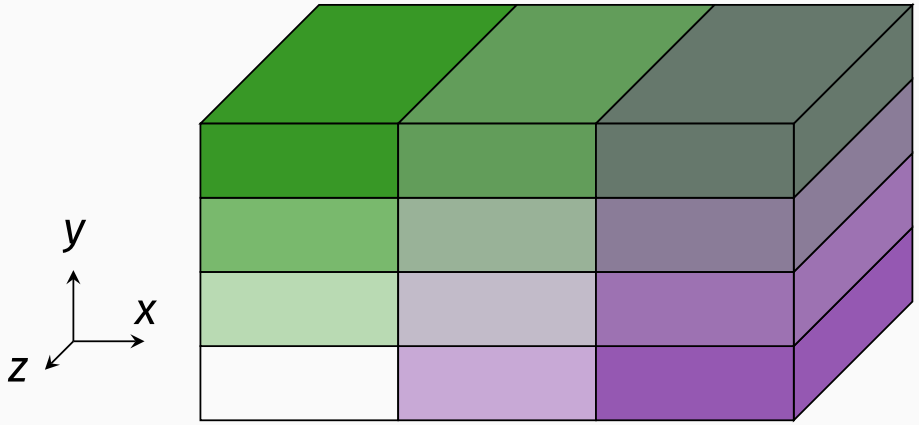
1 brick = 1 MPI process
decomposition along (x, y)
→ 2D (“pencil”) decomposition

$u(x, y, z) \xrightarrow{\text{FFT}(x)} \hat{u}(k_x, y, z)$ is a **global** operation

⇒ For a given (y, z), **all data** must be owned by the same MPI process.

What does this mean for (MPI-style) parallelisation?

Parallelisation based on **domain decomposition**:



1 brick = 1 MPI process

decomposition along (x, y)

→ 2D ("pencil") decomposition

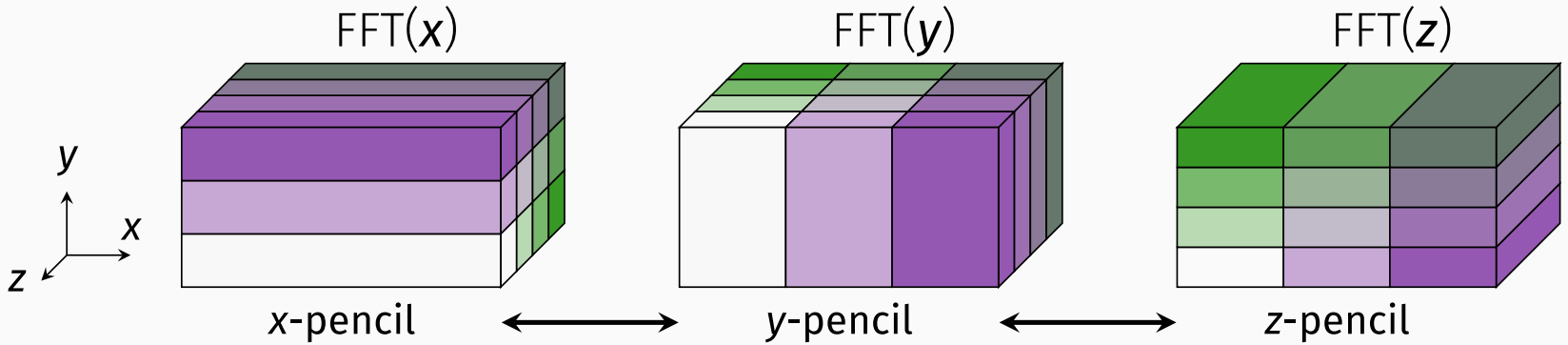
$u(x, y, z) \xrightarrow{\text{FFT}(x)} \hat{u}(k_x, y, z)$ is a **global** operation

⇒ For a given (y, z), **all data** must be owned by the same MPI process.

This is not the case in the above picture!

The “solution”

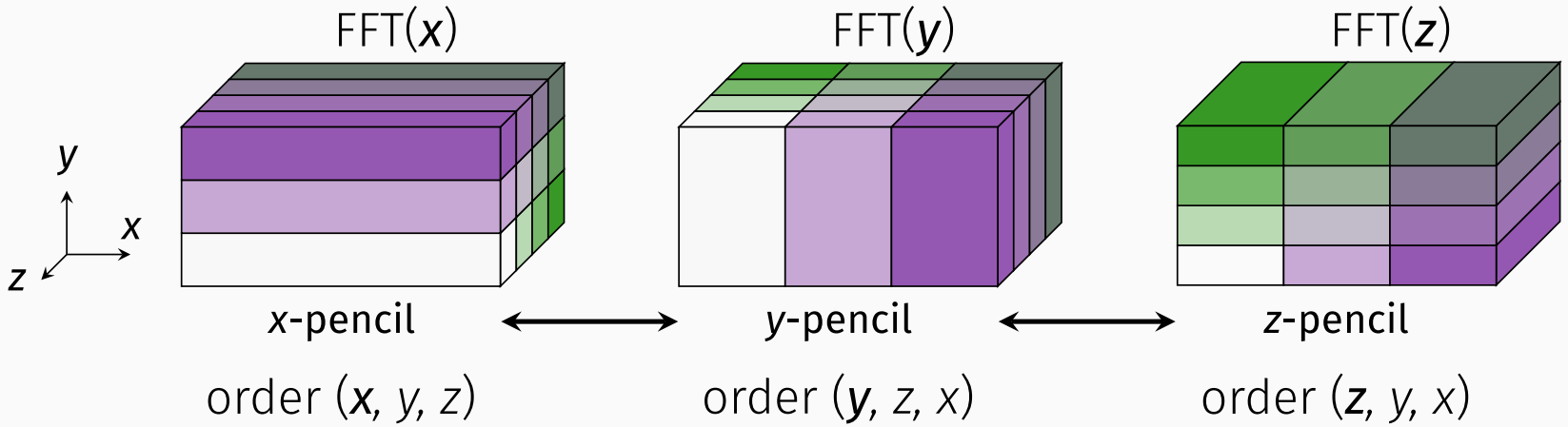
Different decompositions for each 1D transform:



→ requires **transpositions** between configurations (communications!)

The “solution”

Different decompositions for each 1D transform:



→ requires **transpositions** between configurations (communications!)

→ FFT(ζ) is faster if data is **contiguous** along dimension ζ
⇒ output data should be in **different order** than input data

Some existent implementations

Library	Language	Comments
FFTW3[1]	C	1D decomposition only
P3DFFT[2]	Fortran / C++	
2DECOMP&FFT[3]	Fortran	
PFFT[4]	C	arbitrary dimensions
spectralDNS[5]	Python	Navier–Stokes solver
Dedalus[6]	Python	general spectral solver

[1] `http://www.fftw.org/parallel/parallel-fftw.html`
[2] Pekurovsky, *SIAM J. Sci. Comput.* 2012
[3] Li & Laizet, *Cray User Group proceedings* 2010
[4] Pippig, *SIAM J. Sci. Comput.* 2013
[5] Mortensen & Langtangen, *Comput. Phys. Commun.* 2016
[6] Burns, Vasil, Oishi, Lecoanet & Brown, *Phys. Rev. Research* 2020

The PencilArrays.jl package

What does PencilArrays.jl provide?

- easy definition of **domain decomposition** configurations
  (and on **arbitrary** numbers of dimensions)
- `PencilArray` **type** for dealing with distributed data
- efficient **transpositions** between configurations
- zero-cost **index permutations** + contiguous iterators
- convenient **parallel I/O** (both raw MPI-IO and parallel HDF5)
- **distributed FFTs** and related transforms (PencilFFTs.jl package)

A minimal example

script.jl

```julia
using MPI, PencilArrays

MPI.Init()
comm = MPI.COMM_WORLD

Nx, Ny, Nz = (256, 64, 128)  # global domain dimensions

# Automatically generate a decomposition over all processes
pen = Pencil((Nx, Ny, Nz), comm)

# Construct a distributed array
u = PencilArray{Float64}(undef, pen)
```

Running the script

```
> # Create a new Julia project and add required packages to it
> julia --project=. -e 'using Pkg; Pkg.add(["MPI", "PencilArrays"])'
> mpirun -n 12 julia --project script.jl
```

Generating a domain decomposition / the `Pencil` type

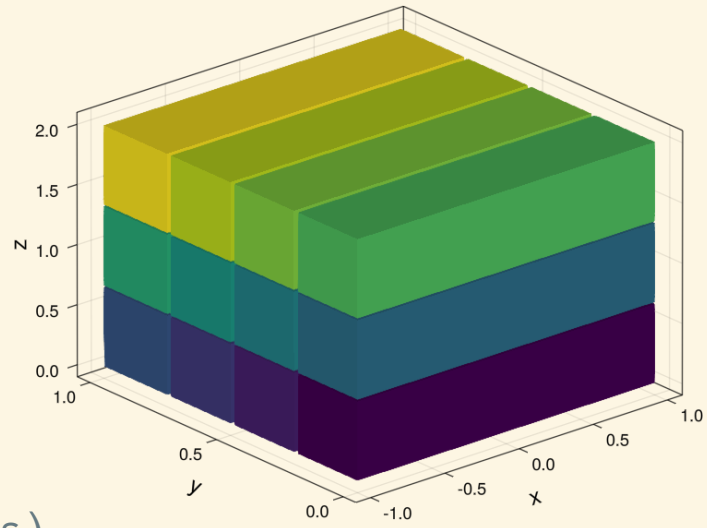
```
julia> Nx, Ny, Nz = (32, 8, 16); comm = MPI.COMM_WORLD;

julia> pen = Pencil((Nx, Ny, Nz), comm)
Decomposition of 3D data
    Data dimensions: (32, 8, 16)
    Decomposed dimensions: (2, 3)
    Data permutation: NoPermutation()
    Array type: Array

julia> topology(pen)
MPI topology: 2D decomposition (4×3 processes)

julia> size_local(pen)  # note: can differ for different ranks!
(32, 2, 5)

julia> range_local(pen)
(1:32, 7:8, 1:5)
```



(Printed outputs are from rank 9 out of 12.)

More control over the decomposition / the `Pencil` type

By default, a 3D domain is decomposed over dimensions `(2,3)`.

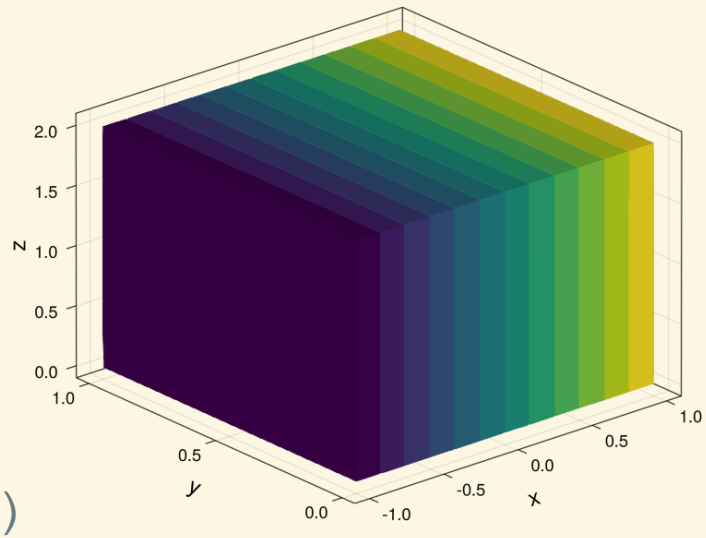
To decompose over dimension `1` instead:

```
julia> dims = (32, 8, 16); comm = MPI.COMM_WORLD;

julia> decomp_dims = (1,);  # decompose over dimension 1 only

julia> pen = Pencil(dims, decomp_dims, comm)
Decomposition of 3D data
    Data dimensions: (32, 8, 16)
    Decomposed dimensions: (1,)
    Data permutation: NoPermutation()
    Array type: Array

julia> topology(pen)
MPI topology: 1D decomposition (12 processes)
```



One can also control the **actual number of processes** in each direction.
See https://jipolanco.github.io/PencilArrays.jl/dev/Pencils/.

```
julia> dims = (32, 8, 16); comm = MPI.COMM_WORLD;
julia> pen = Pencil(dims, comm);

julia> u = PencilArray{Float64}(undef, pen);
julia> rand!(u);  # fill with random values (requires `using Random`)
julia> summary(u)
"32×2×5 PencilArray{Float64, 3}(::Pencil{3, 2, NoPermutation, Array})"

julia> summary(parent(u))  # array actually holding the local data
"32×2×5 Array{Float64, 3}"
```

PencilArray is a simple wrapper type. Roughly:

```
struct PencilArray{T,N} <: AbstractArray{T,N}
    pencil :: Pencil      # domain decomposition information
    data   :: Array{T,N}  # can also be CuArray, etc…
end
```

(The actual implementation is type-stable and a bit more complex…)

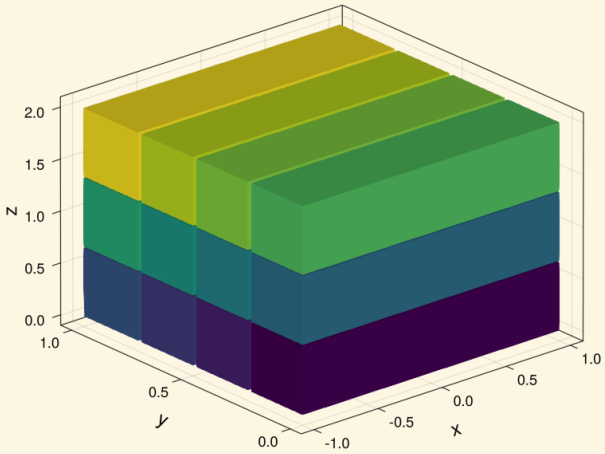
Working with rectilinear grids / the `localgrid` function

```
julia> Nx, Ny, Nz = (33, 9, 17); comm = MPI.COMM_WORLD;
julia> pen = Pencil((Nx, Ny, Nz), comm);

# First define the global grid
julia> xs_global = range(-1, 1; length = Nx);
julia> ys_global = range( 0, 1; length = Ny);
julia> zs_global = range( 0, 2; length = Nz);

# Create a view to the _local_ part of the grid
julia> grid = localgrid(pen, (xs_global, ys_global, zs_global))
LocalRectilinearGrid{3} with coordinates:
 (1) -1.0:0.0625:1.0
 (2) 0.75:0.125:1.0
 (3) 0.0:0.125:0.5

# Initialise a distributed array from local grid values
julia> u = PencilArray{Float64}(undef, pen);
julia> @. u = sin(grid.x) * cos(grid.y) * cos(grid.z);
```



```
julia> grid = localgrid(pen, (xs_global, ys_global, zs_global));
julia> u = PencilArray{Float64}(undef, pen);
julia> @. u = sin(grid.x) * cos(grid.y) * cos(grid.z);
```

Equivalently, using the `grid[i]` syntax:

```
julia> @. u = sin(grid[1]) * cos(grid[2]) * cos(grid[3]);
```

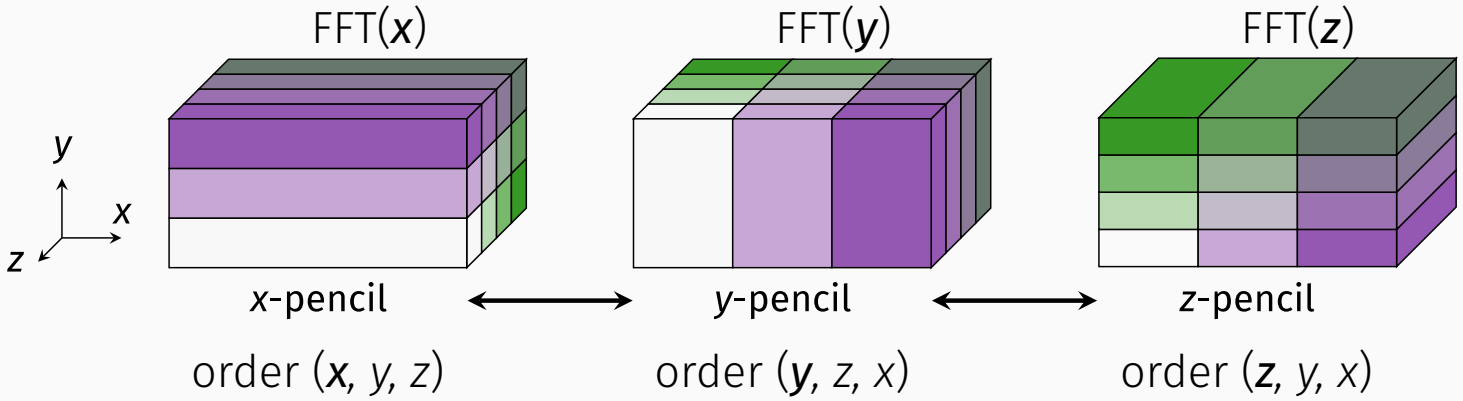
(useful when working with dimensions $N > 3$)

Alternatively, using `for` loops:

```
julia> for I ∈ eachindex(grid)
           x, y, z = grid[I]
           u[I] = sin(x) * cos(y) * cos(z)
       end
```

Dealing with dimension permutations

Dimension permutations

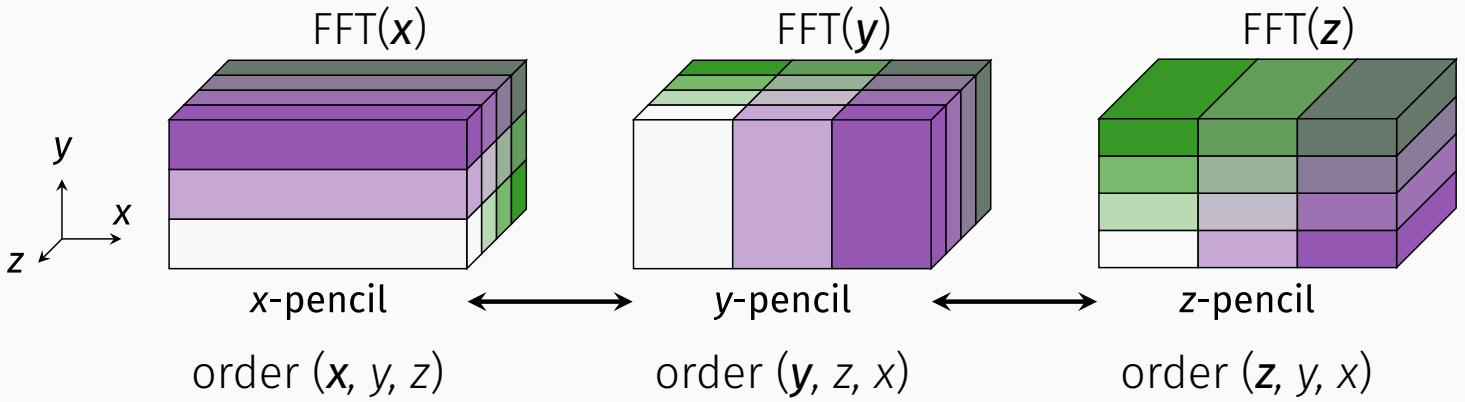


Transform output is **not** in (x, y, z) order!

Possible solutions

- perform **local transposition**: $(z, y, x) \to (x, y, z)$
- work with **permuted indices**: `u[k,j,i]` instead of `u[i,j,k]`
  → inconvenient and potentially confusing
  → difficult to switch between distributed/non-distributed arrays

Dimension permutations



Transform output is **not** in (x, y, z) order!

Possible solutions

- perform **local transposition**: $(z, y, x) \to (x, y, z)$
- work with **permuted indices**: `u[k,j,i]` instead of `u[i,j,k]`
  → inconvenient and potentially confusing
  → difficult to switch between distributed/non-distributed arrays
- use **permutation-aware** array type:
  → internally transform `u[i,j,k]` → `u.data[k,j,i]`

Permutations in PencilArrays.jl

Create decomposition with permutation information:

```
julia> dims = (32, 8, 16); comm = MPI.COMM_WORLD;
julia> pen = Pencil(dims, comm; permute = Permutation(3, 2, 1));
```

Now create a distributed array:

```
julia> u = PencilArray{Float64}(undef, pen);
julia> summary(u)
"32×2×5 PencilArray{Float64, 3}(::Pencil{…})"

julia> summary(parent(u))
"5×2×32 Array{Float64, 3}"  # actual dimensions in memory order

julia> size_local(u)  # or size_local(u, LogicalOrder())
(32, 2, 5)

julia> size_local(u, MemoryOrder())
(5, 2, 32)
```

Indexing distributed arrays

`PencilArray` supports both **linear** and **Cartesian** indexing:

```
julia> pen = Pencil(dims, comm; permute = Permutation(3, 2, 1));
julia> u = PencilArray{Float64}(undef, pen); randn!(u);
julia> size_local(u)
(32, 2, 5)
```

Linear indexing

```
julia> u[42]
-0.866752047759416 9

julia> parent(u)[42]
-0.866752047759416 9
```

Cartesian indexing

```
julia> u[19, 2, 4]
1.5527953306644253

julia> (parent(u))[4, 2, 19]
1.5527953306644253
```

Internally, `(19, 2, 4)` gets swapped to `(4, 2, 19)`.

This has **zero overhead**, since `Permutation(3, 2, 1)` is a compile-time object (StaticPermutations.jl package).

What about loops over permuted arrays?

Regular Julia `Array` — optimal iteration order

```
A = rand(4, 8, 12)
sum = 0.0
for k ∈ axes(A, 3), j ∈ axes(A, 2), i ∈ axes(A, 1)
    c = i + 2j + 3k
    sum += c * A[i, j, k]
end
```

Permuted `PencilArray` — optimal iteration order

```
pen = Pencil(dims, comm; permute = Permutation(3, 2, 1))
A = PencilArray{Float64}(undef, pen)
sum = 0.0
for i ∈ axes(A, 1), j ∈ axes(A, 2), k ∈ axes(A, 3)  # different order!
    c = i + 2j + 3k
    sum += c * A[i, j, k]
end
```

Optimal iteration order **depends on the permutation**??

We can do better...

`CartesianIndices` provides a more generic way:

Regular Julia `Array`

```
A = rand(4, 8, 12)
sum = 0.0
for I ∈ CartesianIndices(A)
    i, j, k = Tuple(I)
    c = i + 2j + 3k
    sum += c * A[I]
end
```

Permuted `PencilArray`

```
A = PencilArray{Float64}(…)
sum = 0.0
for I ∈ CartesianIndices(A)
    i, j, k = Tuple(I)
    c = i + 2j + 3k
    sum += c * A[I]
end
```

Note that `CartesianIndices(::PencilArray)` is overloaded to iterate in **memory order** (i.e. fast!).

If the actual `(i,j,k)` indices are not needed, one can also use `eachindex(A)` or `LinearIndices(A)`.

Summary — dimension permutations

PencilArrays.jl provides a **generic** and **zero-overhead** interface to dimension permutations.

→ allows to **easily switch** between regular and distributed arrays

Somewhat similar to Julia's `PermutedDimsArray` type, but:

- indices are permuted at **compile time**
- iteration is performed in **memory order**

Note that this makes implementing **efficient broadcasting** a bit trickier (but this already works quite well).

Transposing between configurations

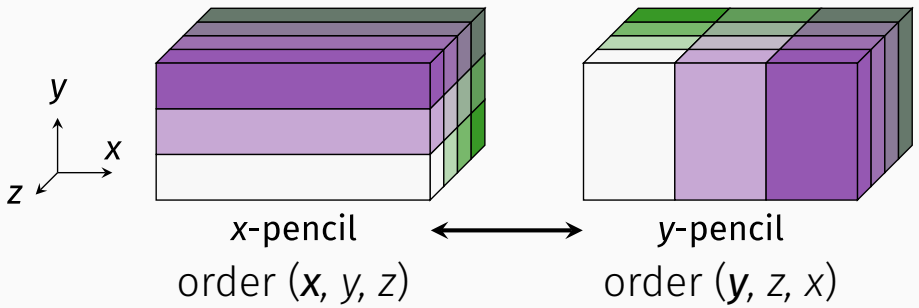
Transposing between configurations / the `transpose!` function

Create 2 *compatible* decomposition configurations:

```
julia> dims = (32, 8, 16); comm = MPI.COMM_WORLD;
julia> pen_x = Pencil(dims, (2, 3), comm);  # decompose along (y, z)
julia> pen_y = Pencil(pen_x; decomp_dims = (1, 3),   #  along (x, z)
                      permute = Permutation(2, 3, 1));
```

Initialise array in "x-pencil" configuration, then transpose to "y-pencil":

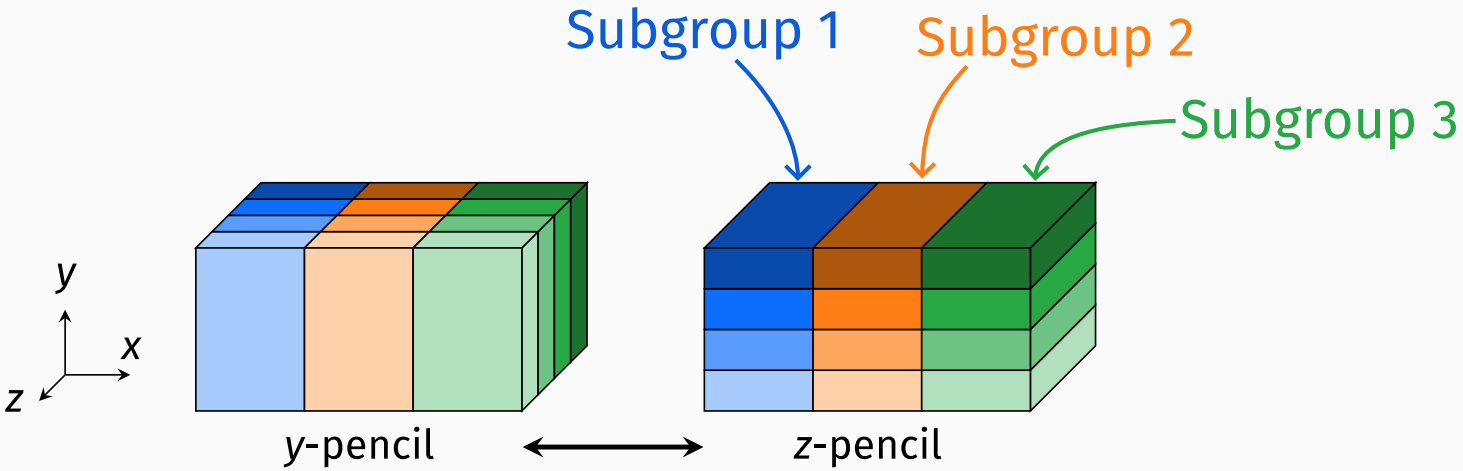
```
julia> ux = PencilArray{Float64}(undef, pen_x); randn!(ux);
julia> uy = PencilArray{Float64}(undef, pen_y);
julia> transpose!(uy, ux);
julia> gather(ux) == gather(uy)  # check that global data is the same
true
```



Transpositions: some implementation details

Communications are performed within **MPI subgroups**.

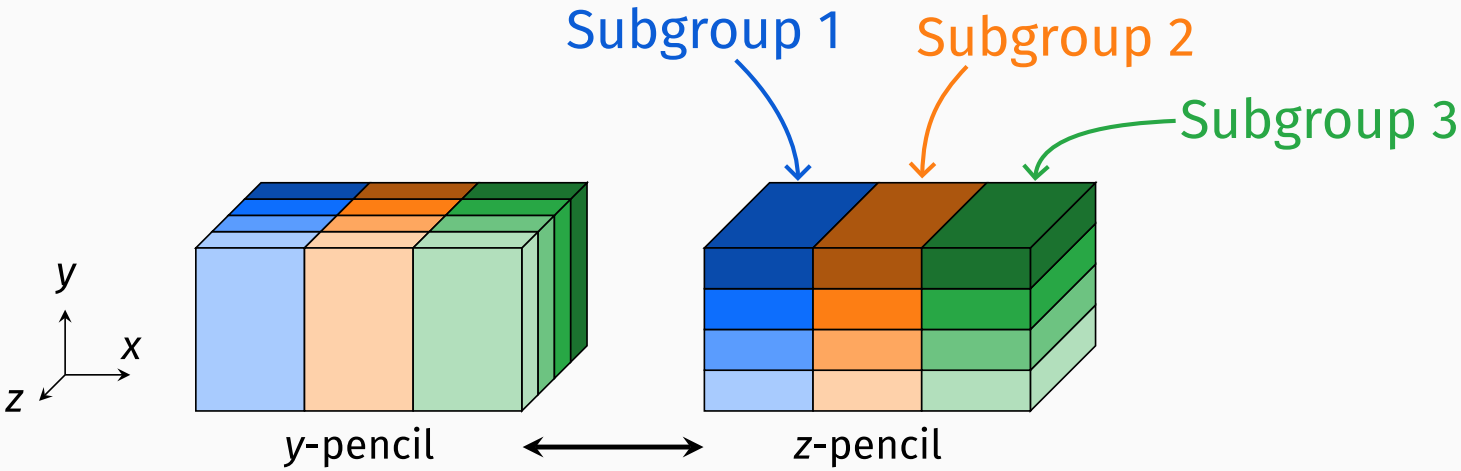
Example: $(1,3) \to (1,2)$ transposition.



Transpositions: some implementation details

Communications are performed within **MPI subgroups**.

Example: `(1,3)` → `(1,2)` transposition.



Two different communication methods:

(1) `method = Transpositions.Alltoallv()`
(2) `method = Transpositions.PointToPoint()` (default)

```
julia> transpose!(uz, uy; method = PointToPoint());
```

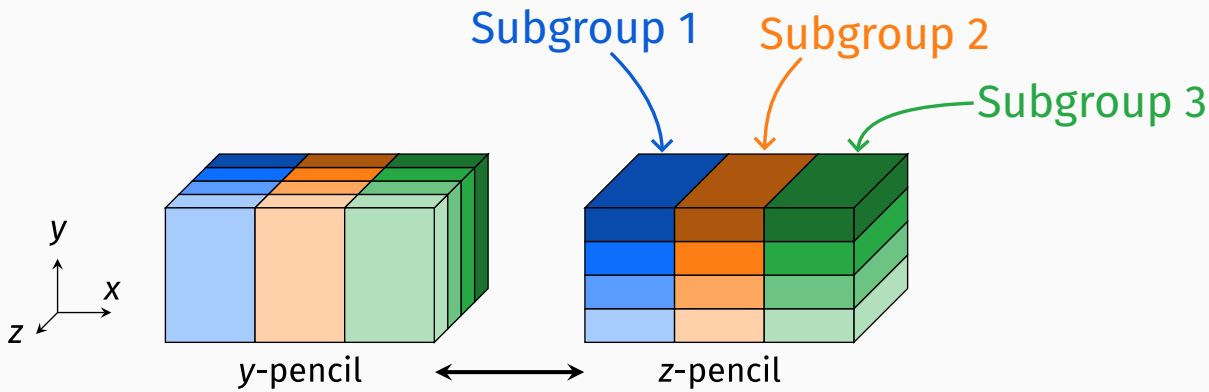
Implemented communication methods

`Traspositions.Alltoallv()`

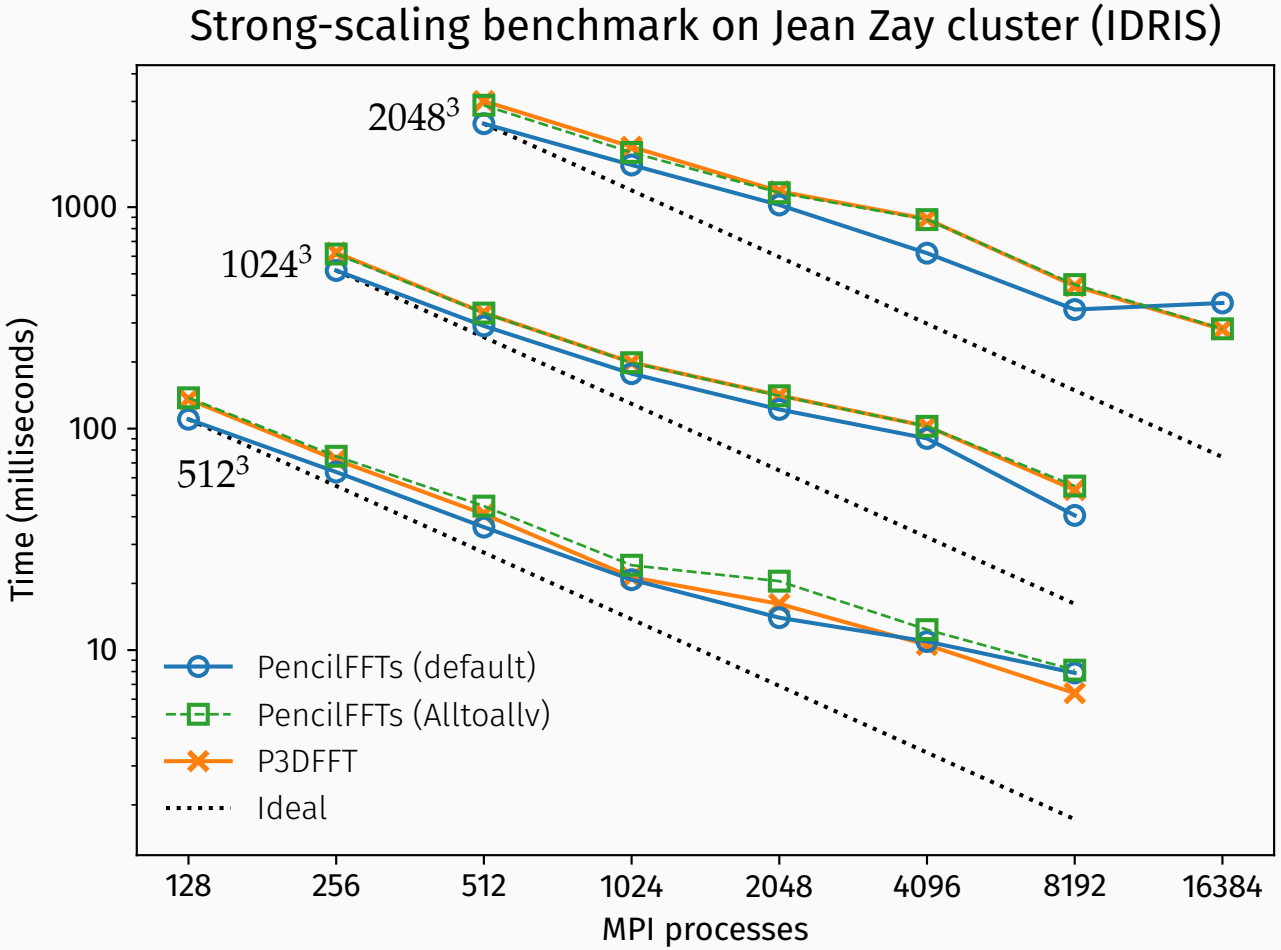
- blocking communication using `MPI_Alltoallv`
- **must wait** before processing any received data

`Transpositions.PointToPoint()`

- fine-grained communication using `MPI_Isend` / `MPI_Irecv`
- can process **partially-received data** while waiting
  → used for local data reordering (permutations)



Performance comparison



Times include transpositions + FFTs.

→ `PointToPoint` **(default)** method makes a difference!

For details: `https://jipolanco.github.io/PencilFFTs.jl/dev/benchmarks/`

Distributed FFTs with PencilFFTs.jl

1. Create distributed array as usual

```
julia> using PencilFFTs, MPI, Random
julia> MPI.Init(); comm = MPI.COMM_WORLD;
julia> pen = Pencil((32, 8, 16), comm);
julia> u = PencilArray{Float64}(undef, pen); randn!(u);
```

2. Create and apply real-to-complex FFT plan

```
julia> plan = PencilFFTPlan(u, Transforms.RFFT())
Transforms: (RFFT, FFT, FFT)
Input type: Float64
Global dimensions: (32, 8, 16) → (17, 8, 16)
MPI topology: 2D decomposition (4×3 processes)

julia> û = plan * u;  # perform distributed FFT
julia> summary(û)
"5×2×16 PencilArray{ComplexF64, 3}(
    ::Pencil{3, 2, Permutation{(3, 2, 1), 3}, Array})"
```

- **all FFTW.jl transforms** are supported
- one can apply **different transforms** along each dimension
  (as long as they're compatible...)

Example (Chebyshev / Fourier / Fourier)

```
using FFTW, PencilFFTs
transforms = (
    Transforms.R2R(FFTW.REDFFT00),  # along x (real-to-real)
    Transforms.RFFT(),              # along y (real-to-complex)
    Transforms.FFT(),               # along z (complex-to-complex)
)
plan = PencilFFTPlan(u, transforms)
```

(PencilFFTs.jl uses serial FFTW transforms behind the scenes)

For details: https://jipolanco.github.io/PencilFFTs.jl/dev/Transforms/

1. Create in-place FFT plan

```
julia> pen = Pencil((32, 8, 16), comm);
julia> plan = PencilFFTPlan(pen, Transforms.FFT!())  # note the `!`
Transforms: (FFT!, FFT!, FFT!)
Input type: ComplexF64
Global dimensions: (32, 8, 16) → (32, 8, 16)
MPI topology: 2D decomposition (4×3 processes)
```

2. Transform array in-place

```
julia> us = allocate_input(plan);    # "ManyPencilArray" container
julia> u = first(us); û = last(us);  # views to input/output
julia> randn!(u);                    # initialise input
julia> plan * us;                    # transform in-place
julia> s = sum(û);                   # operate on the output
```

Here `us` $= (u_1, u_2, u_3)$ is a set of `PencilArrays` which **share memory** while describing **different decomposition configurations**.

Note: in-place **real-to-complex** FFTs are not supported!

A simple example

Example: Laplacian of a periodic scalar field

Consider a scalar field in a $(2\pi)^3$-periodic domain:

$$u(\boldsymbol{x}) = 2\sin(x)\cos(y)\cos(z)$$

$$\Rightarrow \nabla^2 u(\boldsymbol{x}) = -6\sin(x)\cos(y)\cos(z) = -3u(\boldsymbol{x})$$

Some derivatives in Fourier space:

$$\partial_x u(\boldsymbol{x}) \to \mathrm{i}k_x \hat{u}_{\boldsymbol{k}}$$

$$\partial_{xx} u(\boldsymbol{x}) \to -k_x^2 \hat{u}_{\boldsymbol{k}}$$

$$\nabla^2 u(\boldsymbol{x}) \to -|\boldsymbol{k}|^2 \hat{u}_{\boldsymbol{k}} = -\left(k_x^2 + k_y^2 + k_z^2\right)\hat{u}_{\boldsymbol{k}}$$

Steps for computing Laplacian

1.	Define field in physical space	$u(\boldsymbol{x})$
2.	Transform to Fourier space	$\hat{u}_{\boldsymbol{k}}$
3.	Multiply by $-\|\boldsymbol{k}\|^2$	$-\|\boldsymbol{k}\|^2 \hat{u}_{\boldsymbol{k}}$
4.	Transform back to physical space	$\nabla^2 u(\boldsymbol{x})$

Example: Laplacian of a periodic scalar field

1. Define (distributed) field in physical space

```julia
using PencilFFTs, MPI
MPI.Init()
comm = MPI.COMM_WORLD

# Construct distributed array
Nx, Ny, Nz = (32, 32, 32)
pen = Pencil((Nx, Ny, Nz), comm)
u = PencilArray{Float64}(undef, pen)

# Create local physical-space grid
xs_global = range(0, 2π; length = Nx + 1)[1:Nx]
ys_global = range(0, 2π; length = Ny + 1)[1:Ny]
zs_global = range(0, 2π; length = Nz + 1)[1:Nz]
grid = localgrid(pen, (xs_global, ys_global, zs_global))

# Set values of distributed array
@. u = 2 * sin(grid.x) * cos(grid.y) * cos(grid.z)
```

Example: Laplacian of a periodic scalar field

2. Transform to Fourier space

```
plan = PencilFFTPlan(u, Transforms.RFFT())
û = plan * u
```

3. Compute Laplacian (i.e. multiply by $-|\boldsymbol{k}|^2$)

```
# We first need to generate the wavenumber vectors (kx, ky, kz)
using AbstractFFTs: rfftfreq, fftfreq
kx_global = rfftfreq(Nx, Nx)  # = 0, 1, …, Nx/2
ky_global =  fftfreq(Ny, Ny)  # = 0, 1, …, (Ny/2-1), -Ny/2, …, -1
kz_global =  fftfreq(Nz, Nz)  # = 0, 1, …, (Nz/2-1), -Nz/2, …, -1

# Obtain wavenumbers associated to local subdomain
kgrid = localgrid(û, (kx_global, ky_global, kz_global))

# Iterate over the local wavenumber grid, overwriting û
for (I, k⃗) ∈ pairs(kgrid)  # here k⃗ = (kx, ky, kz)
    û[I] *= -(k⃗[1]^2 + k⃗[2]^2 + k⃗[3]^2)
end
# @. û *= -(kgrid[1]^2 + kgrid[2]^2 + kgrid[3]^2)  # this also works
```

Example: Laplacian of a periodic scalar field

4. Transform back to physical space (and verify the result)

```
∇²u = plan \ û
∇²u ≈ -3u  # true!
```

Example: Laplacian of a periodic scalar field

4. Transform back to physical space (and verify the result)

```
∇²u = plan \ û
∇²u ≈ -3u  # true!
```

Side note: using preallocated arrays as transform outputs

As with `FFTW.jl`, one can use `mul!` and `ldiv!` instead of `*` and `\`:

```
julia> using LinearAlgebra: mul!, ldiv!
julia> plan = PencilFFTPlan(u, Transforms.RFFT());
julia> û = allocate_output(plan);
julia> mul!(û, plan, u);   # equivalent to û = plan * u
julia> ldiv!(u, plan, û);  # equivalent to u = plan \ û
```

Parallel I/O

Saving the results to disk

`PencilArrays.jl` comes with **parallel I/O** functionality.

Parallel write using "raw" MPI-IO

```
using PencilArrays.PencilIO  # loads parallel I/O functions
open(MPIIODriver(), "laplacian.bin", comm; write = true) do ff
    ff["scalar"] = u
    ff["scalar_laplacian"] = ∇²u
    ff["scalar_laplacian_fourier"] = û
end
```

→ all processes write to a **single** binary file (`laplacian.bin`)

→ this also writes a JSON file with metadata:

`laplacian.bin.json`

```
{ "driver": {"type": "MPIIODriver", "version": "0.9.4"},
  "datasets": {
    "scalar": { /* stuff… */ }, "scalar_laplacian" : { /* stuff… */ },
    "scalar_laplacian_fourier": {
      "permutation": [3, 2, 1], "little_endian": true, "element_type": "ComplexF64",
      "dims_logical": [17, 32, 32], "chunks": false, "offset_bytes": 524288, /* more stuff… */
}}}
```

There is also a parallel HDF5 driver.

It requires the `HDF5.jl` package built with **parallel HDF5** libraries.

```
using PencilArrays.PencilIO, HDF5
open(PHDF5Driver(), "laplacian.h5", comm; write = true) do ff
    ff["scalar"] = u
    ff["scalar_laplacian"] = ∇²u
    ff["scalar_laplacian_fourier"] = û
end
```

→ all processes write to a **single** HDF5 file (`laplacian.h5`)

```
> h5ls laplacian.h5
scalar                   Dataset {32, 32, 32}
scalar_laplacian         Dataset {32, 32, 32}
scalar_laplacian_fourier Dataset {17, 32, 32}
```

Each dataset includes metadata (e.g. permutations) as HDF5 attributes.

More on parallel I/O

Reading generated files in parallel

```
u = PencilArray{Float64}(undef, pen)
open(MPIIODriver(), "laplacian.bin", comm; read = true) do ff
    read!(ff, u, "scalar")
end
```

Besides, files can be easily read with other tools / languages.

More on parallel I/O

Reading generated files in parallel

```
u = PencilArray{Float64}(undef, pen)
open(MPIIODriver(), "laplacian.bin", comm; read = true) do ff
    read!(ff, u, "scalar")
end
```

Besides, files can be easily read with other tools / languages.

Optional arguments for tuning I/O performance

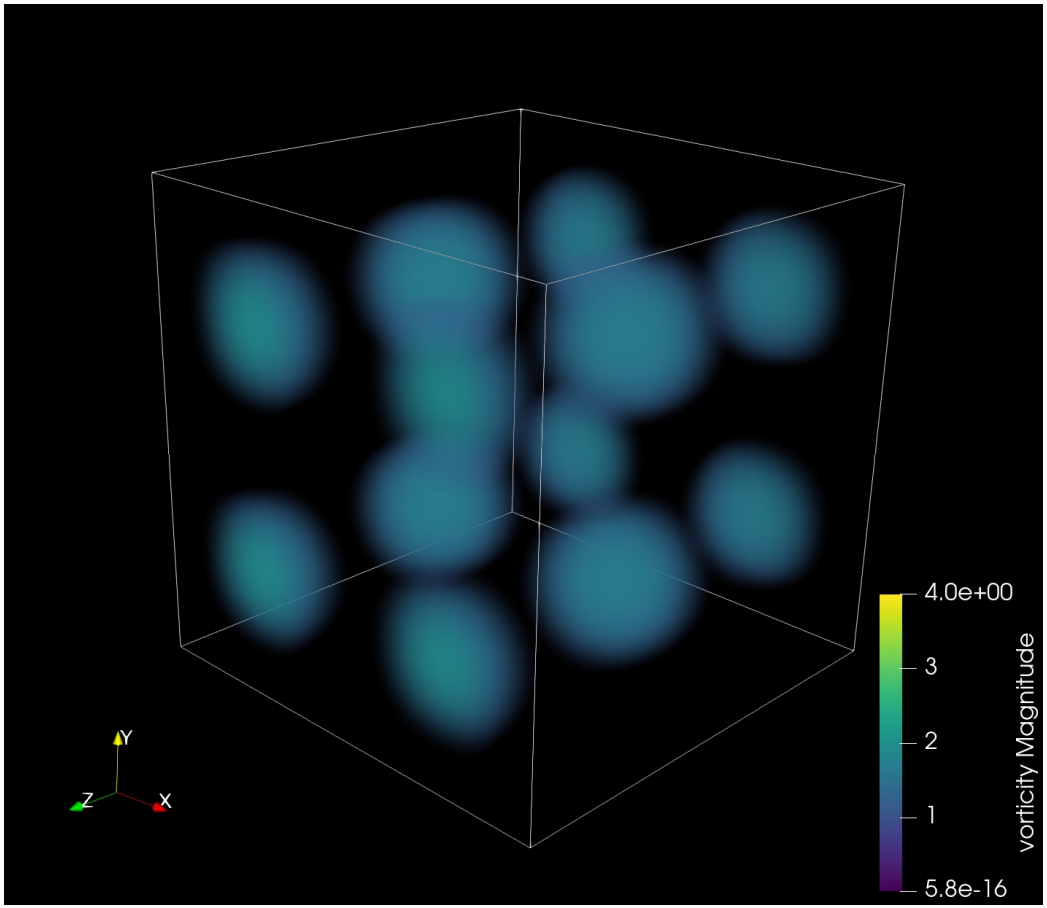
Both drivers support the options:

- `chunks` (default `false`): write one contiguous block per MPI process
  (this has some disadvantages...)
- `collective` (default `true`): perform collective parallel I/O

Example: `ff["scalar", chunks=true, collective=false] = u`

Application: Taylor–Green vortex flow

256^3 **incompressible Navier–Stokes simulation** over 4×4 processes



`OrdinaryDiffEq.jl` used for timestepping (adaptive RK4)

`WriteVTK.jl` + ParaView used for visualisations (from HDF5 output)

Tutorial at `https://jipolanco.github.io/PencilFFTs.jl/dev/generated/navier_stokes/`

Summary and perspectives

Summary

`PencilArrays.jl` provides an **efficient** and **easy to use** interface for solving numerical PDEs on **massively-parallel** computers.

- developed with **spectral methods** in mind, but also useful for other (structured) approaches
- **genericity**: arbitrary N-dimensional geometries can be decomposed over $M < N$ dimensions
- can **interact with other packages** of the Julia ecosystem (e.g. `DifferentialEquations.jl`, `GPUArrays.jl`, …)
- includes **convenient tools** (e.g. for I/O), with more to come…

Perspectives

To be implemented in the **near future**:

- **halo** (or ghost-cell) **exchanges**
  (as in `ImplicitGlobalGrid.jl`, see talk by Ludovic Räss this afternoon)
- improved support for **multi-GPU** configurations
- for visualisation: **XDMF output** along with MPI-IO and HDF5 files
- **pruned FFTs** for better performance when dealiasing
  (implemented e.g. in P3DFFT v2)