Massively Parallel Python Implementation of a Pseudo-Spectral DNS Code for Turbulent Flow

Mikael Mortensen and Hans Petter Langtangen

February 9, 2015

1 Abstract

2 Introduction

Direct Numerical Simulations (DNS) is a term reserved for computer simulations of turbulent flows that are fully resolved in both time and space. DNS are usually conducted using numerical methods of such high quality that numerical dispersion and diffusion errors are negligible compared to their actual physical counterparts. To this end, DNS has historically been carried out with extremely accurate and efficient spectral methods, and in the fluid dynamics community DNS enjoys today the same status as carefully conducted experiments. DNS can provide detailed and highly reliable data not possible to extract from experiments, which in recent years have driven a number of discoveries regarding the very nature of turbulence. The present paper presents a new, computationally attractive tool for performing DNS, realized by recent programming technologies.

Because of the extremely heavy number crunching implied by DNS, researchers aim at highly optimized implementations running on massively parallel computing platforms. The largest known DNS simulations performed today are using hundreds of billions of degrees of freedom. Normally, this demands a need for developing tailored, hand-tuned codes in what we here call low-level languages: Fortran, C or C++. Few DNS codes are openly available and easily accessible to the public and the common fluid mechanics researcher. Some exceptions are hit-3d (Fortran90) [2], Philofluid (Fortran) [4], Tarang (C++) [12], and Turbo (Fortran90) [11]. However, the user interfaces to these codes are not highly developed and it is both challenging and time consuming for a user to to modify or extend the codes to satisfy their own needs. This is usually the nature of low-level codes.

It is a clear trend in computational sciences over the last two decades that researchers tend to move from low-level languages to high-level languages like Matlab, Python, R, and IDL. The experience is that implementations in high-level languages are faster to develop, easier to test, easier to maintain, and they reach a much wider audience because the codes are compact and readable. The downside has been the decreased computational efficiency of high-level languages and in particular their lack of suitability for massively parallel computing. In a field like computational fluid dynamics, this argument has been a show stopper for wide use of high-level languages. However, a language like Python has capabilities today for providing

short and quick implementations that compete with tailored implementations in low-level languages up to thousands of processors. This fact is not well known, and the purpose of this paper is to demonstrate such a result for DNS and show the technical implementation details that are needed.

Python is a language that over the last two decades has grown very popular in the scientific computing community. A wide range of well established, "gold standard" scientific libraries in Fortran and C have been wrapped in Python, making them directly accessible just as commands in MATLAB. There is little overhead in calling low-level Fortran and C/C++functions from Python, and the computational speed obtained in a few lines of code may easily compete with hundreds of compiled lines of Fortran or C code. It is important new knowledge in the CFD community if flow codes can be developed with comfort and ease in Python without sacrificing much computational efficiency.

There are already several examples on successful use of Python for high-performance parallel scientific computing. The sophisticated finite element framework FEniCS (**hpl comment:** *add ref!*) is written mainly in C++, but most application developers are writing FEniCS-based solvers directly in Python, never actually finding themselves in need of writing longer C++ code and firing up a compiler. For large scale applications the devloped Python solvers are usually equally fast as their C++ counterparts, because most of the computing time is usually spent within the low-level wrapped C++ functions that perform the costly linear algebra operations. (**hpl comment:** *Should add parallel FEniCS numbers.*) GPAW (**hpl comment:** *add ref!*) is a code devoted to electronic structure calculations, written as a combination of Python and C. GPAW solvers written in Python have been shown to scale well for thousands of processors. The PETSc project is a major provider of linear algebra to the open source community. PETSc was developed in C, but through the package `petsc4py` almost all routines may be set up and called from Python. PyClaw (**hpl comment:** *add ref!*) is another good example, providing a compact, powerful, and intuitive Python interface to the algorithms within the Fortran codes Clawpack and SharpClaw. PyClaw is parallelised through PETSc and has been shown to scale well up to 65,000 cores.

The ability of Python to wrap low-level, computationally highly efficient Fortran and C/C++ libraries for various applications is today well known, appreciated, and utilized by many. A lesser known fact is that basic Python modules like `numpy`, used for linear algebra and array manipulations, and `mpi4py`, which wraps (nearly) the entire MPI library, may be used directly to develop, from scratch, high performance solvers that run at speeds comparable to the very best implementations in low-level codes. A general misconception seems to be that Python may be used for fast prototyping and post-processing, as MATLAB, but that serious high-performance computing on parallel platforms require reimplementations in Fortran, C or C++. In this paper, we conquer this misconception: The only real requirement for developing a fast pure `numpy`/`mpi4py` solver is that all array manipulations are performed using vectorization (that call underlying BLAS or LAPACK backends) such that explicit for loops over long arrays in Python are avoided. The `mpi4py` module in turn provides a message passing interface for `numpy` arrays at communication speeds very close to pure C code.

The major objective of this work is to explain a novel implementation of an excellent research tool for DNS, aimed at a wide audience. To this end, we i) show how a complete pseudo-spectral DNS solver can be written from scratch in Python using less than 100 lines

of compact, very readable code, and ii) show that these 100 lines of code can run at speeds comparable to its low-level counterpart in hand-written C++ code. To establish scaling and benchmark results, we have run the codes on SHAHEEN, a massively parallel BlueGene/P machine at the KAUST Supercomputing Laboratory.

3 The Navier-Stokes equations in spectral space

Our DNS implementation is based on a pseudo-spectral Galerkin method [1] for the spatial discretization. The Navier-Stokes equations are first cast in rotational form

$$\frac{\partial \boldsymbol{u}}{\partial t} - \boldsymbol{u} \times \boldsymbol{\omega} = \nu \nabla^2 \boldsymbol{u} - \nabla P, \tag{1}$$

$$\nabla \cdot \boldsymbol{u} = 0, \tag{2}$$

$$\boldsymbol{u}(\boldsymbol{x} + 2\pi \boldsymbol{e}^i, t) = \boldsymbol{u}(\boldsymbol{x}, t), \quad \text{for } i = 1, 2, 3, \tag{3}$$

$$\boldsymbol{u}(\boldsymbol{x}, 0) = \boldsymbol{u}_0(\boldsymbol{x}) \tag{4}$$

where $\boldsymbol{u}(\boldsymbol{x}, t)$ is the velocity vector, $\boldsymbol{\omega} = \nabla \times \boldsymbol{u}$ the vorticity vector, $\boldsymbol{e}^i$ the Cartesian unit vectors, and the modified pressure $P = p + \boldsymbol{u} \cdot \boldsymbol{u}/2$, where p is the regular pressure normalized by the constant density. The equations are periodic in all three spatial directions. If all three directions now are discretized uniformely in space using a structured computational mesh with N points in each direction, the mesh points can be represented as

$$x_j = \frac{2\pi j}{N}, \quad j = 0, \ldots, N-1, \tag{5}$$

$$\boldsymbol{x}_j^i = x_j \boldsymbol{e}^i, \quad j = 0, \ldots, N-1, \ i = 1, 2, 3. \tag{6}$$

All variables may be transformed from the physical mesh $\boldsymbol{x}_j^i$ to a discrete and bounded Fourier wavenumber mesh using three-dimensional discrete Fourier transforms. Each point in the physical mesh will take part in three consecutive transformations, one for each periodic direction. The first transformed direction (arbitrary which one) is real and the remaining two are complex valued. There are in total N^2 real transforms of length N and $2(N/2+1)N$ complex transforms of length N. The transforms in the three directions are performed sequentially. The first real transform along one line in the z-direction reads

$$\mathcal{F}_{k_z}(\boldsymbol{u}) = \hat{\boldsymbol{u}}_{k_z}(t) = \frac{2\pi}{N} \sum_{j=0}^{N-1} \boldsymbol{u}(x_j, t) e^{-ik_z x_j}, \quad k_z = -N/2 + 1, \ldots, N/2, \tag{7}$$

with the inverse transform

$$\mathcal{F}_j^{-1}(\hat{\boldsymbol{u}}) = \boldsymbol{u}(x_j, t) = \frac{1}{2\pi} \sum_{k_z=-N/2+1}^{N/2} \hat{\boldsymbol{u}}_{k_z}(t) e^{ik_z x_j}, \quad j = 0, \ldots, N-1. \tag{8}$$

Note that i is used both as a spatial counter and as the imaginary unit. The meaning should be clear from the context.

The transform is performed along all N^2 lines on the cubic mesh in the z-direction. To simplify notation, the complete three-dimensional transform for the entire mesh used in this work is denoted as

$$\mathcal{F}_{\boldsymbol{k}}(\boldsymbol{u}) = \hat{\boldsymbol{u}}_{\boldsymbol{k}}(t) = \mathcal{F}_{k_x}\left(\mathcal{F}_{k_y}\left(\mathcal{F}_{k_z}(\boldsymbol{u})\right)\right), \quad \boldsymbol{k} = (k_x, k_y, k_z). \tag{9}$$

Note, however, that the order is arbitrary except from the data layout in memory. Similarily, using i, j, k for the three Cartesian directions x, y, z, the inverse transform is defined as

$$\mathcal{F}^{-1}_{\boldsymbol{x}}(\hat{\boldsymbol{u}}) = \boldsymbol{u}(\boldsymbol{x}, t) = \mathcal{F}^{-1}_{k}\left(\mathcal{F}^{-1}_{j}\left(\mathcal{F}^{-1}_{i}(\hat{\boldsymbol{u}})\right)\right), \quad \boldsymbol{x} = (x, y, z), \tag{10}$$

where the inverse transforms are taken in the opposite order of the forward transforms.

By taking the Fourier transform of the Navier-Stokes equations and subsequently the analythical spatial derivatives in spectral space, we obtain a system of ordinary differential equation for $\hat{\boldsymbol{u}}_{\boldsymbol{k}}$, and the continuity equation reduces to an orthogonal inner product:

$$\frac{d\hat{\boldsymbol{u}}_{\boldsymbol{k}}}{dt} - \widehat{(\boldsymbol{u} \times \boldsymbol{\omega})}_{\boldsymbol{k}} = -\nu|\boldsymbol{k}|^2\hat{\boldsymbol{u}}_{\boldsymbol{k}} - i\boldsymbol{k}\hat{P}_{\boldsymbol{k}}, \tag{11}$$

$$i\boldsymbol{k} \cdot \hat{\boldsymbol{u}}_{\boldsymbol{k}} = 0. \tag{12}$$

The pressure may be eliminated by taking the divergence of (1), or equivalently by dotting the transformed (11) by $i\boldsymbol{k}$ and rearranging such that

$$\hat{P}_{\boldsymbol{k}} = -i\frac{\boldsymbol{k} \cdot \widehat{(\boldsymbol{u} \times \boldsymbol{\omega})}_{\boldsymbol{k}}}{|\boldsymbol{k}|^2}. \tag{13}$$

Inserting for the pressure in (11), the final equation to solve for each transformed velocity vector is thus

$$\frac{d\hat{\boldsymbol{u}}_{\boldsymbol{k}}}{dt} = \widehat{(\boldsymbol{u} \times \boldsymbol{\omega})}_{\boldsymbol{k}} - \nu|\boldsymbol{k}|^2\hat{\boldsymbol{u}}_{\boldsymbol{k}} - \boldsymbol{k}\frac{\boldsymbol{k} \cdot \widehat{(\boldsymbol{u} \times \boldsymbol{\omega})}_{\boldsymbol{k}}}{|\boldsymbol{k}|^2}. \tag{14}$$

Note that the transformed velocity components are coupled through the nonlinear convection term and the eliminated pressure.

The pseudo-spectral label arises from the treatment of the convective term, which is computed by first transforming the velocity and vorticity to physical space, performing the cross product, and then transforming the vector $(\boldsymbol{u} \times \boldsymbol{\omega})$ back to Fourier space. The operation requires 2 inverse transforms (velocity and vorticity) and 1 forward transform for each of the three vector components, 9 all together. Note that this is the only operation that requires MPI communication and it is typically the most computationally extensive part of a pseudo-spectral DNS solver.

The time integration of (14) is performed explicitly using a fourth-order Runge-Kutta method, the Forward Euler method or a second-order Adams-Bashforth method. The details are left out here, but all algorithms simply need a function that returns the right hand side of (14).

4 Implementation

We have implemented the pseudo-spectral discretization of the Navier-Stokes equations, as described in the previous section, in high-level Python code. It is strongly remarked that this Python code is not simply a wrapper of a low-level, high-performance flow solver in Fortran, C or C++. The entire code is implemented directly in Python: the mesh, the MPI domain decomposition, and the time integrators. We are only making use of wrappers for FFT, something that is also done by the majority of low-level flow solvers anyway. The current Python implementation may, as such, be used as a working prototype for a complete low-level implementation in Fortran, C or C++.

The Python solver makes extensive use of the `numpy` and `mpi4py` packages, but if the `pyfftw` module has been installed, it will be used to perform the FFT instead of `numpy.fft`. (**hpl comment:** *With so heavy use of code as part of the text, I found it inconvenient to have it in figures. It's like factorizing out all the equations in the text in separate figures... The code floats around and suddenly the text discusses a code, but the nearby figure with code relates to something two pages back. Now all code is inline.*)

Python code

```
from numpy import *
from numpy.fft import fftfreq, fft, ifft, rfft, irfft
try:
    from mpi.wrappyfftw import *
except ImportError:
    pass # Rely on numpy.fft routines
from mpi4py import MPI
comm = MPI.COMM_WORLD
num_processes = comm.Get_size()
rank = comm.Get_rank()
```

Importing `MPI` from `mpi4py` initializes the MPI communicator. Two different strategies, slab and pencil, have been implemented for the MPI parallel domain decomposition. However, since communication only enters through the FFTs, there is very little difference between a serial code and a parallel one. Therefore, we first present a serial version of the code.

4.1 Serial version of code

The computational mesh is in physical space a structured uniform cube $[0, 2\pi]^3$, where each direction is divided into N uniform intervals, where $N = 2^M$ for a postive integer M. Any different size of the box may be easily implemented through scaling. The mesh according to (6) is represented in Python as

Python code

```
M = 6
N = 2**M
X = mgrid[:N, :N, :N].astype(float)*L/N
```

The matrix `X` has dimensions `(3, N, N, N)`. Since the Navier Stokes equations are solved in Fourier space, the physical space is only used to compute the convection plus to do post

processing. The mesh `X` is typically used for initialization and otherwise not needed (and may therefore be deleted to save memory). In parallel mode, `X` will be split up and divided between the processors. Note that the solver may be operated in either single or double precision mode, and that `float` in our code is a placeholder for either one of the `numpy` datatypes `float32` or `float64`, depending on settings.

The velocity field to be transformed is real, and the discrete Fourier transform of a real sequence has the property that $\hat{\boldsymbol{u}}_k = \hat{\boldsymbol{u}}^*_{N-k}$, where * denotes the complex conjugate. As such, it is sufficient to use $N/2+1$ Fourier coefficients, leading to a transformed wavenumber mesh of size $(N/2+1)N^2$. The highest frequency (the Nyquist frequency $k = N/2+1$) is often neglected in turbulence simulations because "this is a coefficient for a Fourier mode that is not carried in the Fourier representation of the solution" [5]. For the MPI slab decomposition the Nyquist frequency is included, whereas it is neglected for the 2D pencil decomposition (**hpl comment:** *Why?*). For MPI communication reasons, the real transform is taken in the final z direction and the wavenumbers $\boldsymbol{k} = (k_x, k_y, k_z)$ stored on the transformed mesh thus has ordering as used by the FFT routines provided by `numpy.fft` or `pyfftw`:

$$\begin{aligned}\boldsymbol{k} = [&(0, \ldots, N/2-1, -N/2, -N/2+1, \ldots, -1),\\ &(0, \ldots, N/2-1, -N/2, -N/2+1, \ldots, -1),\\ &(0, \ldots, N/2-1, N/2)].\end{aligned} \tag{15}$$

The wavenumber mesh is implemented as

Python code

```
Nf = N/2+1
kx = ky = fftfreq(N, 1./N).astype(int)
kz = kx[:Nf].copy(); kz[-1] *= -1
K = array(meshgrid(kx, ky, kz, indexing='ij'), dtype=int)
K2 = sum(K*K, 0, dtype=int)
K_over_K2 = K.astype(float) / where(K2 == 0, 1, K2).astype(float)
```

where `fftfreq(N,1./N)` is a function that creates the wavenumbers $(0, \ldots, N/2-1, -N/2, -N/2+1, \ldots, -1)$. The dimesions of the matrices are `(3, N, N/2+1, N)` for `K`, `(N, N/2+1, N)` for `K2` and `(3, N, N/2+1, N)` for `K_over_K2`, and these matrices represent $\boldsymbol{k}$, $|\boldsymbol{k}|^2$ and $\boldsymbol{k}/|\boldsymbol{k}|^2$, respectively. The last two matrices are precomputed for efficiency.

The velocity, curl and pressure are similarily stored in structured numpy arrays

Python code

```
U     = empty((3, N, N, N),  dtype=float)
U_hat = empty((3, N, N, Nf), dtype=complex)
P     = empty((N, N, N),     dtype=float)
P_hat = empty((N, N, Nf),    dtype=complex)
curl  = empty((3, N, N, N))
```

Here `hat` denotes a transformed variable. To transform between, e.g., `U` and `U_hat`, calls to FFT routines are required. The three dimensional FFT and its inverse are implemended in Python functions as shown below.

Python code

```
def fftn_mpi(u, fu):
    """FFT of u in three directions."""
    if num_processes == 1:                  # Scalar version
        fu[:] = rfftn(u, axes=(0,1,2))
    return fu

def ifftn_mpi(fu, u):
    """Inverse FFT of fu in three directions."""
    if num_processes == 1:                  # Scalar version
        u[:] = irfftn(fu, axes=(0,1,2))
    return u

# Usage
U_hat = fftn_mpi(U, U_hat)
U = ifftn_mpi(U_hat, U)
```

For high performance, it is key that the Python code relies on *in-place* modifications of pre-allocated arrays to avoid unnecessary allocation of large temporary arrays (which often arises from basic `numpy` code). Each of the functions above takes the result array (`U_hat` or `U`) as argument, fill this array with values and returns the array to the calling code. A commonly applied convention in Python is to return all result objects from functions as this only involves transfer of references and no copying of data.

We also remark that the three consecutive transforms performed by `rfftn/irfftn` are actually using one real transform along the y-direction and two complex transforms in the remaining two directions. Also note that the simple `numpy/pyfftw` wrapped functions `rfftn/irfftn` may only be used in single processor mode, and the MPI implementation is detailed in Sections 4.2 and 4.3.

The convection term requires a transform from Fourier to physical space where the cross product $\boldsymbol{u} \times \boldsymbol{\omega}$ is carried out. The curl in Fourier space is

$$\mathcal{F}_{\boldsymbol{k}}(\nabla \times \boldsymbol{u}) = \hat{\boldsymbol{\omega}}_{\boldsymbol{k}} = i\boldsymbol{k} \times \hat{\boldsymbol{u}}_{\boldsymbol{k}}. \tag{16}$$

We can now compute the curl in physical space through $\boldsymbol{\omega} = \mathcal{F}_{\boldsymbol{x}}^{-1}(\hat{\boldsymbol{\omega}})$. The convection term may thus be computed as

$$\widehat{(\boldsymbol{u} \times \boldsymbol{\omega})}_{\boldsymbol{k}} = \mathcal{F}_{\boldsymbol{k}}(\boldsymbol{u} \times \boldsymbol{\omega}) = \mathcal{F}_{\boldsymbol{k}}(\mathcal{F}_{\boldsymbol{x}}^{-1}(\hat{\boldsymbol{u}}) \times \mathcal{F}_{\boldsymbol{x}}^{-1}(\hat{\boldsymbol{\omega}})). \tag{17}$$

The Python functions for the curl and cross products are

Python code

```
def Cross(a, b, c):
    c[0] = fftn_mpi(a[1]*b[2]-a[2]*b[1], c[0])
    c[1] = fftn_mpi(a[2]*b[0]-a[0]*b[2], c[1])
    c[2] = fftn_mpi(a[0]*b[1]-a[1]*b[0], c[2])
    return c

def Curl(a, c):
    c[2] = ifftn_mpi(1j*(K[0]*a[1]-K[1]*a[0]), c[2])
    c[1] = ifftn_mpi(1j*(K[2]*a[0]-K[0]*a[2]), c[1])
```

```
    c[0] = ifftn_mpi(1j*(K[1]*a[2]-K[2]*a[1]), c[0])
    return c
```

The entire right hand side of (14) is implemented as shown in Fig. **??**.

The convection term in (14) is dealiased using the 2/3-rule and dealiasing is simply achieved by an elementwise multiplication of a convection matrix `dU[3, N, N/2+1, N]` with a matrix `dealias[N,N/2+1,N]`, which is zero where the wavenumbers are larger than 2/3 of the Nyquist mode and unity otherwise. Note that the dimensions of `dU` and `dealias` differ in the first index since `dU` contains contributions for all three vector components. However, through automatic broadcasting, `numpy` realizes that the last three dimensions are the same and as such all three components of `dU` (i.e., `dU[0]`, `dU[1]` and `dU[2]`) are multiplied elementwise with the same matrix `dealias`.

Python code

```
# Declare some matrices and parameters
kmax_dealias = N/3.
dU = empty((3, N, Nf, N), dtype=complex)  # Holds right-hand side
dealias = array((abs(K[0]) < kmax_dealias)*(abs(K[1]) < kmax_dealias)*
                (abs(K[2]) < kmax_dealias), dtype=bool)
dt = 0.01     # Time step
nu = 0.001    # Viscosity

def ComputeRHS(dU, rk):
    if rk > 0: # For rk=0 the correct values are already in U
        for i in range(3):
            U[i] = ifftn_mpi(U_hat[i], U[i])

    # Compute convective term and place in dU
    curl = Curl(U_hat, curl)
    dU = Cross(U, curl, dU)

    # Dealias the nonlinear convection
    dU *= dealias*dt

    # Compute pressure (to get actual pressure multiply by 1j/dt)
    P_hat[:] = sum(dU*K_over_K2, 0)

    # Add pressure gradient
    dU -= P_hat*K

    # Add viscous term
    dU -= nu*dt*K2*U_hat

    return dU
```

(**hpl comment:** *Remark: for in-place operators it is more common to write* `a += b` *than* `a[:] += b`*, but the result is the same.*) (**hpl comment:** `dt` *should be taken out of the rhs routine.*) (**hpl comment:** *Is not* $\frac{1}{2}u^2$ *part of the* P *pressure here too?*)

The simple Forward Euler integrator is now

Python code

```
t = 0          # Physical time
T = 1.0        # End time
while t <= T:
    t += dt
    U_hat[:] += ComputeRHS(dU, 0)
```

4.2 1D Slab decomposition

To run the code on several processors using MPI, the data structures need to be split up. The most popular strategy in the litterature is the "slab" decomposition, where each CPU is assigned responsibility for a certain number of complete 2D planes (slices). In other words, just one of the three indices (i, j, k) is split up and divided amongst the CPUs. The major drawback of the slab decomposition strategy is that the number of CPUs must be smaller than or equal to N for a cubic mesh of size N^3.

The meshes for the slab decomposition are implemented as

Python code

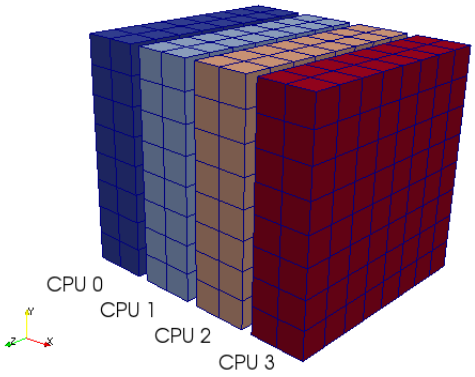
```
Np = N / num_processes
X = array(meshgrid(x[rank*Np:(rank+1)*Np], x, x, indexing='ij'))
K = array(meshgrid(kx, ky, kz[rank*Np:(rank+1)*Np],
                   indexing='ij'), dtype=int)
```

In general, using `num_processes` CPUs, each CPU gets responsibility for `Np = N/num_processes` slices and the physical mesh is simply decomposed along the first index. The wavenumber mesh, on the other hand, is split along the third direction. The reason for this choice is that the k_x direction is the last direction to be transformed by the three consecutive FFTs and for this operation the data structures need to be aligned in the k_x direction.

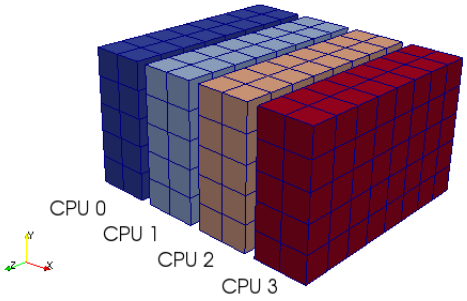
The MPI decomposition is illustrated in Fig. 1 for the case of a physical mesh of size 8^3 divided among 4 CPUs. Each processor performs a complete two dimensional FFT in both y and z directions on the original real data structure. To do the final transform in the x-direction, data must be communicated between all processors. The communication takes place in one single MPI `Alltoall` operation and one transform from the data structure in Fig 2(b) to the one in 2(c). The entire 3D parallel FFT may be implemented with 5 lines of Python code:

Python code

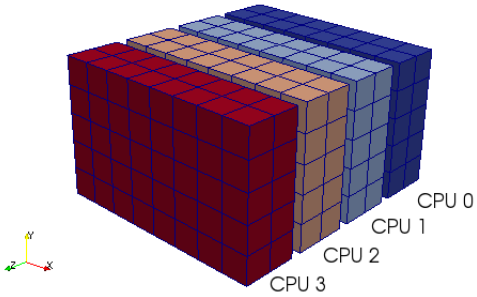
```
def fftn_mpi(u, fu):
    """FFT in three directions using MPI."""
    Uc_hatT[:] = rfft2(u, axes=(2,1))
    for i in range(num_processes):
        U_mpi[i] = Uc_hatT[:, :, i*Np:(i+1)*Np]
    comm.Alltoall([U_mpi, MPI.DOUBLE_COMPLEX], [fu, MPI.DOUBLE_COMPLEX])
    fu[:] = fft(fu, axis=0)
    return fu

def ifftn_mpi(fu, u):
    """Inverse FFT in three directions using MPI.
       Need to do ifft in reversed order of fft."""
```



(a)Physical mesh



(b)Wavenumber mesh after real transform



(c)Final wavenumber mesh.

Figure 1: Slab decomposition of physical (a) and wavenumber meshes (b), (c).

```
    Uc_hat[:] = ifft(fu, axis=0)
    comm.Alltoall([Uc_hat, mpitype], [U_mpi, mpitype])
    for i in range(num_processes):
        Uc_hatT[:, :, i*Np:(i+1)*Np] = U_mpi[i]
    u[:] = irfft2(Uc_hatT, axes=(2,1))
    return u
```

Three global pre-allocated complex work arrays are needed: `Uc_hatT[Np,N/2+1,N]`, `Uc_hatT[N,N/2+1,Np]` and `Uc_mpi[N,N/2+1,Np]`. Note that `Alltoall` requires two work arrays as placeholders, one for each of the data structures seen in Fig 1 (b) and (c). Alternatively, the data transfer may be performed without explicit work arrays using the in-place `Sendrecv_replace` along with transpose operations, as shown in alternative versions of these functions in the next section.

4.3 2D Pencil decomposition

For massively parallel simulations the slab decomposition falls short since the number of CPUs allowed is limited by N. Large-scale simulations using up to N^2 CPUs commonly employ the 2D pencil decomposition, first suggested by Ding, Ferraro and Gennery in 1995 [3], is commonly used. Publically available implementations of the 3D parallel FFT that makes use of the pencil decomposition are the Parallel FFT Subroutine Library by [10], the P3DFFT library by Pekurovsky [7, 8], the 2DECOMP&FFT library by Li and Laizet [6] and PFFT by Pippig [9].

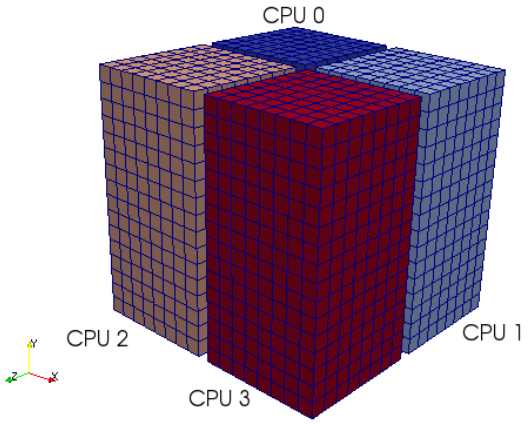
The 2D pencil decomposition strategy is illustrated in Fig. 2 for a box of size 16^3, using 4 CPUs. The datastructures are split in the plane normal to the direction where we are performing the 1D FFTs. That is, for the physical mesh in Fig. 2 (a) the $x-z$ plane is split up in a 2×2 processor mesh. Each CPU may now perform 64 ($= 8 \times 8$) 1D real FFTs in the y-direction on its $8 \times 8 \times 16$ physical mesh. Afterwords, the complex data are laid out as shown in Fig. 2 (b). The second transform takes place in the x-direction, and for this to happen, data must be exchanged between processors 0 and 1 as well as 2 and 3. The datastructures must also be transposed to the shape seen in Fig. 2 (c). Each CPU may now perform the 32 (4×8) individual 1D FFTs in the x-direction. The same procedure is followed to end up with datastructures aligned in the final z-direction, see Fig. 2 (d). However, this time processor 0 communicates with processor 2 and likewise 1 with 3.

Evidently, each processor belongs to two different groups of communicators. One for communication in the $x-y$ plane (Fig. 2 (b) to (c)) and one for communication in the $x-z$ plane (Fig 2 (c) to (d)). The MPI communicator groups and the distributed mesh are created in Python as shown below.

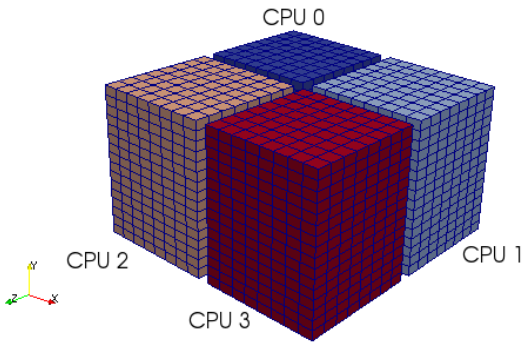
Python code

```
num_processes = comm.Get_size()  # Total number of CPUs
rank = comm.Get_rank()           # Global rank
P1 = 2                           # CPUs in 1st direction
P2 = num_processes / P1          # CPUs in 2nd direction
N1 = N/P1
N2 = N/P2

# Create two communicator groups for each rank
commxz = comm.Split(rank%P1)
commxy = comm.Split(rank/P1)

xyrank = commxy.Get_rank() # Local rank in xy-plane
xzrank = commxz.Get_rank() # Local rank in xz-plane

# Create the decomposed physical mesh X
x = linspace(0, L, N+1).astype(float)[:-1]
x1 = slice(xyrank * N1, (xyrank+1) * N1, 1)
x2 = slice(xzrank * N2, (xzrank+1) * N2, 1)
X = array(meshgrid(x[x1], x, x[x2], indexing='ij'), dtype=float)
```

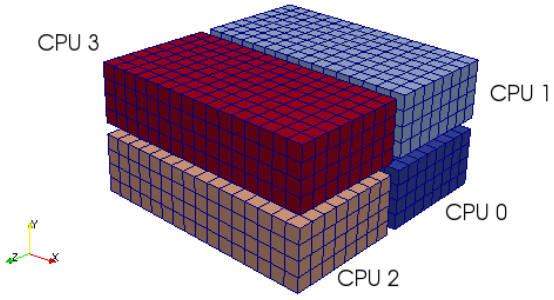
With reference to Fig. 2, the two communicator groups for CPU with global rank 0 is `commxy` = [0, 1] and `commxz` = [0, 2].



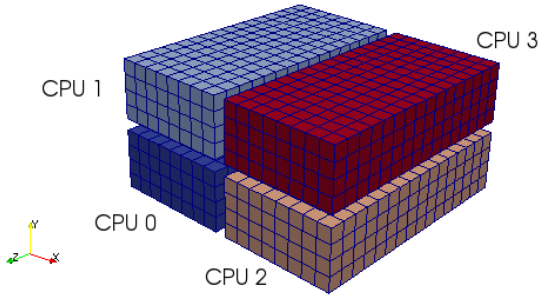
(a)Physical mesh



(b)Wavenumber mesh after real transform



(c)Intermediate wavenumber mesh



(d)Final wavenumber mesh.

Figure 2: 2D pencil decomposition of physical mesh (a) and the three wavenumber meshes (b), (c), (d). The decomposition shown uses 4 CPUs, two in each direction normal to the direction of the current one-dimensional FFT. The FFT in the y-direction transforms data in (a) to (b). The data is then transformed and communicated to the composition seen in (c). Here the FFT in x-direction is carried out before the data is transformed again and communicated to (d), where the final FFT is performed.

The entire 9 lines of code for the Python implementation of the 3D parallel FFT with the 2D pencil decomposition appear below. The work arrays `Uc_hat_y,Uc_hat_x,Uc_hat_z` are laid out as seen in Fig. 2 (b), (c) and (d) respectively. The array `Uc_hat_xr` is a copy of `Uc_hat_x` used only for communication.

Python code

```
def fftn_mpi(u, fu):
    """Three dimensional FFT using MPI."""
    # Do FFT in y direction on owned real data u
    Uc_hat_y[:] = rfft(u, axis=1)

    # Transform to x direction neglecting Nyquist mode
    for i in range(P1):
        Uc_hat_x[i*N1:(i+1)*N1] = Uc_hat_y[:, i*N1/2:(i+1)*N1/2]

    # Communicate and do FFT in x-direction
    commxy.Alltoall([Uc_hat_x, mpitype], [Uc_hat_xr, mpitype])
    Uc_hat_x[:] = fft(Uc_hat_xr, axis=0)

    # Communicate and transform to final z-direction
    commxz.Alltoall([Uc_hat_x, mpitype], [Uc_hat_xr, mpitype])
    for i in range(P2):
        Uc_hat_z[:, :, i*N2:(i+1)*N2] = Uc_hat_xr[i*N2:(i+1)*N2]

    # Do FFT for last direction
    fu[:] = fft(Uc_hat_z, axis=2)
    return fu
```

Note that data communication is performed when the data are aligned with the x-direction (Fig. 2 (c)). Here the datastructure `Uc_hat_x` is of shape `(N, N1/2, N2)`, where `N1=N/P1` and `N2=N/P2` and `P1` and `P2` are the division of processors used for the two split directions. In Fig. 2 `P1` $=$ `P2` $= 2$. In the call to `Alltoall` the first dimension of `Uc_hat_x`, i.e. N, is automatically broadcasted such that `Uc_hat_x` gets the shape `(P1, N1, N1/2, N2)`.

Python code

```
def fftn_mpi(u, fu):
    """FFT in 3D using an alternative MPI implementation."""
    # Do FFT for two directions, first real
    ft = fu.transpose(2,1,0)
    ft[:] = rfft2(u, axes=(2,1))

    # Create transformed view of fu array
    fu_send = fu.reshape((num_processes, Np, Nf, Np))

    # Communicate
    for i in range(num_processes):
        if not i == rank:
            comm.Sendrecv_replace([fu_send[i], MPI.DOUBLE_COMPLEX], i, 0,
                i, 0)

    # Align data with final x-direction
    fu_send[:] = fu_send.transpose(0,3,2,1)
```

```
    # Do FFT for last direction
    fu[:] = fft(fu, axis=0)
    return fu

def ifftn_mpi(fu, u):
    """ifft in 3D using an alternative MPI implementation."""
    # Do inverse FFT of first owned direction
    Uc_hat[:] = ifft(fu, axis=0)

    # Create transformed view of Uc_hat
    Uc_send = Uc_hat.reshape((num_processes, Np, Nf, Np))

    # Communicate and transform
    for i in range(num_processes):
        if not i == rank:
            comm.Sendrecv_replace([Uc_send[i],
                                   MPI.DOUBLE_COMPLEX], i, 0, i, 0)
        Uc_hatT[:, :, i*Np:(i+1)*Np] = Uc_send[i]

    # Do inverse FFT for last two directions, final real
    u[:] = irfft2(Uc_hatT, axes=(2,1))
    return u
```

Note that no explicit work arrays are used with the forward transform.

References

[1] Claudio Canuto. *Spectral Methods in Fluid Dynamics (Scientific Computation)*. Springer-Verlag New York-Heidelberg-Berlin, 1987.

[2] S. G Chumakov. https://code.google.com/p/hit3d/.

[3] Hong Q. Ding, Robert D. Ferraro, and Donald B. Gennery. A portable 3d FFT package for distributed-memory parallel architectures. In *PPSC*, pages 70–71, 1995.

[4] M. et. al. Iovieno. http://areeweb.polito.it/ricerca/philofluid/.

[5] Myoungkyu Lee, Nicholas Malaya, and Robert D. Moser. Petascale direct numerical simulation of turbulent channel flow on up to 786k cores. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 61:1–61:11, New York, NY, USA, 2013. ACM.

[6] N. Li and S. Laizet. 2DECOMP&FFT – A highly scalable 2D decomposition library and FFT interface. In *Cray User Group conference, Edinburgh*. 2010.

[7] D. Pekurovsky. P3DFFT. http://code.google.com/p/p3dfft/.

[8] D. Pekurovsky. P3DFFT: a framework for parallel computations of Fourier transforms in three dimensions. *SIAM Journal on Scientific Computing*, 34(4), 2012.

[9] Michael Pippig. PFFT - An extension of FFTW to massively parallel architectures. *SIAM J. Sci. Comput.*, 35:C213 – C236, 2013.

[10] S. J. Plimpton. Parallel FFT Subroutine Library. http://www.sandia.gov/∼sjplimp/docs/fft/README.html.

[11] B. et. al. Teaca. http://aqua.ulb.ac.be/home/turbo/.

[12] M Vermi. http://turbulence.phy.iitk.ac.in/doku.php.