

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) [5], *Philofluid* (Fortran) [8], *Tarang* (C++) [19], and *Turbo* (Fortran90) [18]. 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, C or 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 [1] 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 developed Python solvers are usually equally fast as their C++ counterparts, because most of the computing time typically is spent within the low-level wrapped C++ functions that perform the costly linear algebra operations [13]. GPAW [12] 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 [3] is a major provider of linear algebra to the open source community. PETSc was developed in C, but through the package `petsc4py` [6] almost all routines may be set up and called from Python. PyClaw [9] 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` DNS 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, 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 Navier-Stokes in spectral space

The DNS implementation uses a pseudo-spectral Galerkin method [4]. The Navier-Stokes equations are first cast in rotational form

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

$$\nabla \cdot \mathbf{u} = 0, \quad (2)$$

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

$$\mathbf{u}(\mathbf{x}, 0) = \mathbf{u}_0(\mathbf{x}) \quad (4)$$

where $\mathbf{u}(\mathbf{x}, t)$ is the velocity vector, $\boldsymbol{\omega} = \nabla \times \mathbf{u}$ the vorticity vector, \mathbf{e}^i the Cartesian unit vectors, and the modified pressure $P = p + \mathbf{u} \cdot \mathbf{u}/2$, 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 uniformly in space using a structured computational mesh with N points in each direction, the mesh may be represented as

$$x_j = \frac{2\pi j}{N}, \quad j = 0, \dots, N-1, \quad (5)$$

$$\mathbf{x}_j^i = x_j \mathbf{e}^i, \quad j = 0, \dots, N-1, \quad i = 1, 2, 3. \quad (6)$$

All variables may be transformed from the real mesh \mathbf{x}_j^i to a discrete and bounded Fourier wavenumber mesh using three-dimensional discrete Fourier transforms. Each point in the real 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. Altogether there are 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

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

with the inverse transform

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

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

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

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

$$\mathcal{F}_{\mathbf{x}}^{-1}(\hat{\mathbf{u}}) = \mathbf{u}(\mathbf{x}, t) = \mathcal{F}_k^{-1} \left(\mathcal{F}_j^{-1} \left(\mathcal{F}_i^{-1}(\hat{\mathbf{u}}) \right) \right), \quad \mathbf{x} = (x, y, z), \quad (10)$$

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 analytical spatial derivatives in spectral space, we obtain an ordinary differential equation for $\hat{\mathbf{u}}_{\mathbf{k}}$ and the continuity equation reduces to an orthogonal inner product

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

$$i\mathbf{k} \cdot \hat{\mathbf{u}}_{\mathbf{k}} = 0. \quad (12)$$

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

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

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

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

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 real space, performing the cross product, and then transforming the vector $(\mathbf{u} \times \boldsymbol{\omega})$ 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 DNS solver.

The time integration of (14) is performed explicitly using a fourth order Runge-Kutta method, the 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).

```

from numpy import *
from numpy.fft import fftfreq, fft, ifft, rfft, irfft
from mpi4py import MPI
comm = MPI.COMM_WORLD
num_processes = comm.Get_size()
rank = comm.Get_rank()

```

Figure 1: First lines of the solver are responsible for importing required functionality and for initializing the MPI communicator.

4 Implementation

A Python solver for the pseudo-spectral Navier-Stokes equations is created in one main executable module. It is important to realize that the Python solver is not simply a wrapper of a low-level high-performance solver. Everything is implemented directly in Python - the mesh, the MPI domain decomposition and the Runge-Kutta integrator. We are only making use of wrappers for FFT, something that is also done by the majority of low-level solvers anyway. The current Python implementation may, as such, be used as a working prototype for a low-level implementation.

The solver makes use of the `numpy` and `mpi4py` packages that are imported from in the very first part of the solver, see Fig. 1. Note that if the `pyfftw` module [2] has been installed, it should be used to perform the FFT instead of `numpy`. Importing MPI from `mpi4py` initializes the MPI communicator. Two different strategies, slab and pencil, have been implemented for the MPI domain decomposition. However, since communication only enters through the FFTs there is very little difference between a serial code and a parallel, and thus we first present a serial version of the code.

4.1 Serial version of code

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

```

M = 6      # Assign the size of the mesh
N = 2**M   # Compute size of the mesh in each direction
X = mgrid[:N, :N, :N].astype(float)*L/N

```

where the matrix `X` has dimensions `X[3, N, N, N]`. Since the Navier Stokes equations are solved in Fourier space, the real space is only used to compute the convection plus to do postprocessing. The mesh `X` is typically used for initialization and otherwise not needed. As such, it may 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` 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 $\hat{\mathbf{u}}_k = \hat{\mathbf{u}}_{N-k}^*$, 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" [10]. For the MPI slab decomposition the Nyquist frequency is included, whereas it is neglected for the 2D pencil decomposition. For MPI communication reasons, the real transform is taken in the final z direction and the wavenumbers $\mathbf{k} = (k_x, k_y, k_z)$ stored on the transformed mesh thus has ordering as used by the FFT routines provided by numpy (and pyfftw)

$$\mathbf{k} = [(0, \dots, N/2 - 1, -N/2, -N/2 + 1, \dots, -1), \\ (0, \dots, N/2 - 1, -N/2, -N/2 + 1, \dots, -1), \\ (0, \dots, N/2 - 1, N/2)]. \quad (15)$$

The wavenumber mesh is implemented as

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

where `fftfreq(N, 1./N)` is a function that creates the wavenumbers $(0, \dots, N/2-1, -N/2, -N/2+1, \dots, -1)$. The dimensions of the matrices are $KX[3, N, N/2+1, N]$, $KK[N, N/2+1, N]$ and $KX_over_Ksq[3, N, N/2+1, N]$, and the matrices represent \mathbf{k} , $|\mathbf{k}|^2$ and $\mathbf{k}/|\mathbf{k}|^2$ respectively. The last two matrices are precomputed for efficiency.

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

```
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` represents a transformed variable. To tranform between, e.g., \mathbf{U} and $\mathbf{U_hat}$, the FFTs are required. The three dimensional FFT and its inverse are implemented in Python functions as shown in Fig. (2). Note 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 Sec ?.

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

$$\mathcal{F}_k(\nabla \times \mathbf{u}) = \hat{\boldsymbol{\omega}}_k = i\mathbf{k} \times \hat{\mathbf{u}}_k. \quad (16)$$

```

def fftn_mpi(u, fu):
    """fft of u in three directions. Result in fu.
    """
    if num_processes == 1:
        fu[:] = rfftn(u, axes=(0,1,2))

def ifftn_mpi(fu, u):
    """ifft of fu in three directions. Result in u.
    """
    if num_processes == 1:
        u[:] = irfftn(fu, axes=(0,1,2))

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

```

Figure 2: Serial implementation of three dimensional FFT and inverse FFT.

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

$$(\widehat{\mathbf{u} \times \boldsymbol{\omega}})_k = \mathcal{F}_k(\mathbf{u} \times \boldsymbol{\omega}) = \mathcal{F}_k(\mathcal{F}_x^{-1}(\hat{\mathbf{u}}) \times \mathcal{F}_x^{-1}(\hat{\boldsymbol{\omega}})). \quad (17)$$

where the implemented functions to do the curl and cross products are

```

def Cross(a, b, c):
    """c = F(a x b)"""
    fftn_mpi(a[1]*b[2]-a[2]*b[1], c[0])
    fftn_mpi(a[2]*b[0]-a[0]*b[2], c[1])
    fftn_mpi(a[0]*b[1]-a[1]*b[0], c[2])

def Curl(a, c):
    """c = F_inv(curl(a))"""
    ifftn_mpi(1j*(KX[0]*a[1]-KX[1]*a[0]), c[2])
    ifftn_mpi(1j*(KX[2]*a[0]-KX[0]*a[2]), c[1])
    ifftn_mpi(1j*(KX[1]*a[2]-KX[2]*a[1]), c[0])

```

The entire right hand side of (14) is implemented as shown in Fig. 3. The convection term is dealiased using the 2/3-rule and dealiasing is simply achieved by an elementwise multiplication of the convection matrix $\mathbf{dU}[3, N, N/2+1, N]$ with matrix $\mathbf{dealias}[N, N/2+1, N]$, that is zero where the wavenumbers are larger than 2/3 of the Nyquist mode and one otherwise. Note that the dimensions of \mathbf{dU} and $\mathbf{dealias}$ differ in the first index since \mathbf{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 \mathbf{dU} (i.e., $\mathbf{dU}[0]$, $\mathbf{dU}[1]$ and $\mathbf{dU}[2]$) are multiplied elementwise with the same matrix $\mathbf{dealias}$. The parameters ν and Δt are the viscosity and time step respectively and they are both declared as global variables in the beginning of the solver. The advancement in time can be performed in its simplest possible form through the Forward Euler method as shown in Fig. 4.

```

# Declare some matrices and parameters
kmax = N/3.
dU = empty((3, N, Nf, N), dtype=complex)
dealias = array((abs(KX[0]) < kmax)*(abs(KX[1]) < kmax)*
                (abs(KX[2]) < kmax), 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):
            ifftn_mpi(U_hat[i], U[i])

    # Compute convective term and place in dU
    Curl(U_hat, curl)
    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*KX_over_Ksq, 0)

    # Add pressure gradient
    dU[:] -= P_hat*KX

    # Add contribution from diffusion
    dU[:] -= nu*dt*KK*U_hat

```

Figure 3: Function for computing the right hand side of Eq. (14).

4.2 1D Slab decomposition

To run the code on several processors using MPI, the data structures need to be split up. In the literature the most popular strategy is "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 shown in Fig. 5. In general, using `num_processes` CPUs, each CPU gets responsibility for $N_p = N/\text{num_processes}$ 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 (see Fig. 2) and for this operation the data structures need to be aligned in the k_x direction.

The MPI decomposition is easily understood by the illustrations in Fig. 6, which is using 4 CPUs for a physical mesh of size 8^3 . Each processor does 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


```

t = 0          # Actual time
T = 1.0        # End time
while t < T-1e-8:
    t += dt
    timestep += 1
    ComputeRHS(dU, 0)
    U_hat[:, :] += dU

```

Figure 4: Forward Euler method for time advancement of Eq. (14).

```

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

```

Figure 5: 1D slab decomposition of physical mesh and wavenumber mesh.

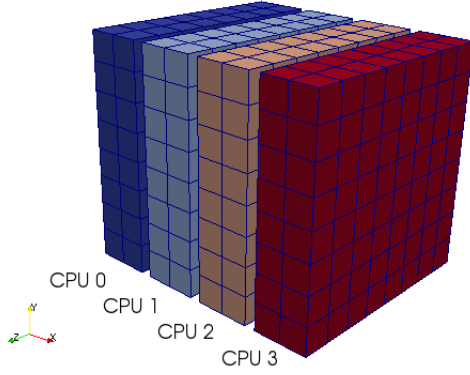
place in one single MPI `Alltoall` operation and one transform from the data structure in Fig 8(b) to the one in 8(c). The entire 3D parallel FFT may be implemented with 5 lines of Python code as shown in Fig 7. Note that `Alltoall` requires two work arrays as placeholders, one for each of the data structures seen in Fig 6 (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 the commented code in Fig 11.

4.3 2D Pencil decomposition

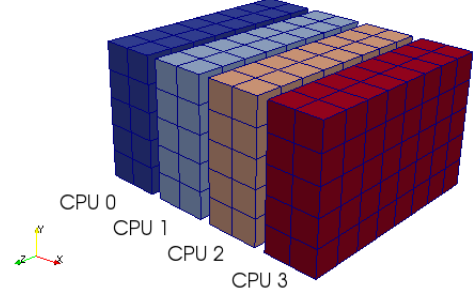
For massively parallel simulations the slab decomposition falls short since the number of CPUs allowed is limited by N , where the size of the physical box is N^3 . For large scale simulations using up to N^2 CPUs, the 2D pencil decomposition, first suggested by Ding, Ferraro and Gennerly in 1995 [7], 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 [17], the P3DFFT library by Pekurovsky [14, 15], the 2DECOMP&FFT library by Li and Laizet [11] and PFFT by Pippig [16].

The 2D pencil decomposition strategy is illustrated in Fig. 8 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 real mesh in Fig. 8 (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 $8 \times 8 \times 16$ real mesh. Afterwards, the complex data will be laid out as shown in Fig. 8 (b). The second transform will take 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. 8 (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. 8 (d). However, this time processor 0 communicates with processor 2 and likewise 1 with 3.

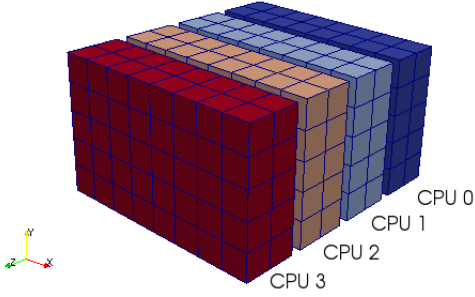
Evidently, each processor belongs to two different groups of communicators. One for



(a)Physical mesh



(b)Wavenumber mesh after real transform



(c)Final wavenumber mesh.

Figure 6: Slab decomposition of physical (a) and wavenumber meshes (b), (c).

communication in the $x - y$ plane (Fig. 8 (b) to (c)) and one for communication in the $x - z$ plane (Fig 8 (c) to (d)). The MPI communicator groups and the distributed mesh are created in Python as shown in Fig. 9.

The entire 9 lines of code for the Python implementation of the 3D parallel FFT is shown in Fig. 10. Note that data communication is performed when the data are aligned with the x -direction (Fig. 8 (c)). Here the datastructure is of shape $Uc_hat_x[N, N1/2, N2]$, where $N1=N/P0$ and $N2=N/P1$ and $P0$ and $P1$ are the division of processors used for the two split directions. In Fig. 8 $P0 = P1 = 2$. In the call to `Alltoall` the first dimension of Uc_hat_x , i.e. N , is automatically broadcasted such that $Uc_hat_x[P1, N1, N1/2, N2]$

References

- [1] FEniCS.

```

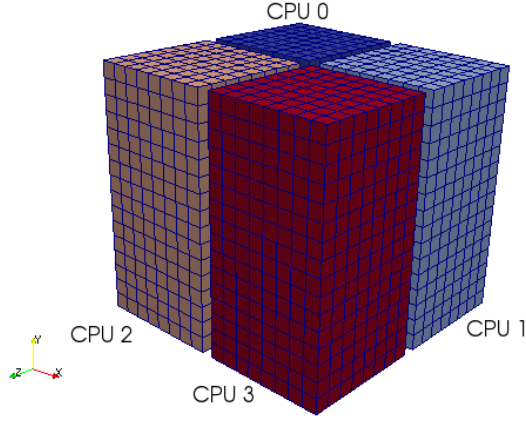
def fftn_mpi(u, fu):
    """fft in three directions using mpi
    """
    Uc_hatT[:] = rfft2(u, axes=(2,1))
    for i in range(num_processes):
        U_mpi[i] = Uc_hatT[:, :, i*Np:(i+1)*Np]
    comm.Alltoall([U_mpi, MPI.DOUBLE_COMPLEX], [fu, MPI.DOUBLE_COMPLEX])
    fu[:] = fft(fu, axis=0)

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

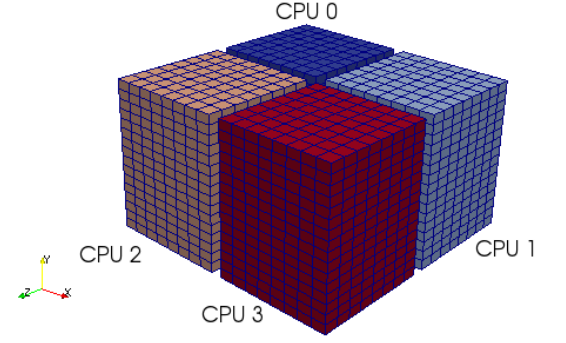
```

Figure 7: Three dimensional parallel FFT and inverse FFT with slab decomposition. Three intermediate complex work arrays $Uc_hatT[Np, N/2+1, N]$, $Uc_hatT[N, N/2+1, Np]$ and $Uc_mpi[N, N/2+1, Np]$ are used.

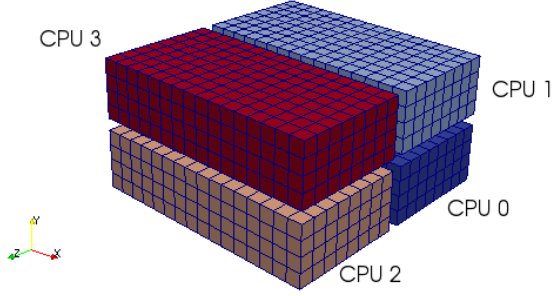
- [2] <https://pypi.python.org/pypi/pyfftw>.
- [3] Satish Balay, Jed Brown, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2013. <http://www.mcs.anl.gov/petsc>.
- [4] Claudio Canuto. *Spectral Methods in Fluid Dynamics (Scientific Computation)*. Springer-Verlag New York-Heidelberg-Berlin, 1987.
- [5] S. G Chumakov. <https://code.google.com/p/hit3d/>.
- [6] Lisandrop Dalcin. <https://bitbucket.org/petsc/petsc4py>.
- [7] 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.
- [8] M. et. al. Iovieno. <http://areeweb.polito.it/ricerca/philofluid/>.
- [9] D. Ketcheson, K. Mandli, A. Ahmadi, A. Alghamdi, M. de Luna, M. Parsani, M. Knepley, and M. Emmett. Pyclaw: Accessible, extensible, scalable tools for wave propagation problems. *SIAM Journal on Scientific Computing*, 34(4):C210–C231, 2012.
- [10] 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.



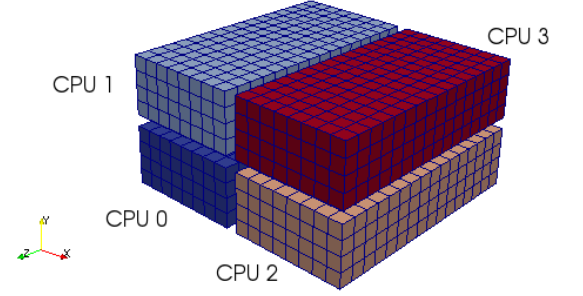
(a) Physical mesh



(b) Wavenumber mesh after real transform



(c) Intermediate wavenumber mesh



(d) Final wavenumber mesh.

Figure 8: 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.

```

num_processes = comm.Get_size() # Total number of CPUs
rank = comm.Get_rank() # Global rank
P1 = 2 # Assign number of CPUs in first direction
P2 = num_processes / P1 # CPUs in second 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 physical mesh
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)

```

Figure 9: Creation of two MPI communicator groups, `commxy` and `commxz`, and the decomposed physical mesh `X[3, N1, N, N2]`. With reference to Fig. 8, the two communicator groups for CPU with global rank 0 is `commxy = [0, 1]` and `commxz = [0, 2]`.

```

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)

```

Figure 10: Three dimensional FFT with pencil decomposition. The work arrays `Uc_hat_y`, `Uc_hat_x`, `Uc_hat_z` are laid out as seen in Fig. 8 (b), (c) and (d) respectively. The array `Uc_hat_xr` is a copy of `Uc_hat_x` used only for communication.

```

def fftn_mpi(u, fu):
    """fft in three directions using mpi
    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)

def ifftn_mpi(fu, u):
    """ifft in three directions using mpi.
    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))

```

Figure 11: Three dimensional parallel FFT with slab decomposition. Note that no explicit work arrays are used with the forward transform.

- [11] N. Li and S. Laizet. 2DECOMP&FFT – A highly scalable 2D decomposition library and FFT interface. In *Cray User Group conference, Edinburgh*. 2010.
- [12] J. J. Mortensen, L. B. Hansen, and K. W. Jacobsen. Real-space grid implementation of the projector augmented wave method. *Phys. Rev. B*, 71(3):035109, JAN 2005.
- [13] Mikael Mortensen and Kristian Valen-Sendstad. Oasis: A high-level/high-performance open source Navier-Stokes solver. *Computer Physics Communications*, 188(0):177 – 188, 2015.
- [14] D. Pekurovsky. P3DFFT. <http://code.google.com/p/p3dfft/>.
- [15] D. Pekurovsky. P3DFFT: a framework for parallel computations of Fourier transforms in three dimensions. *SIAM Journal on Scientific Computing*, 34(4), 2012.
- [16] Michael Pippig. PFFT - An extension of FFTW to massively parallel architectures. *SIAM J. Sci. Comput.*, 35:C213 – C236, 2013.
- [17] S. J. Plimpton. Parallel FFT Subroutine Library. <http://www.sandia.gov/~sjplimp/docs/fft/README.html>.
- [18] B. et. al. Teaca. <http://aqua.ulb.ac.be/home/turbo/>.
- [19] M Vermi. <http://turbulence.phy.iitk.ac.in/doku.php>.