Computer Physics Communications 203 (2016) 53–65

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

High performance Python for direct numerical simulations of turbulent flows

Mikael Mortensen *, Hans Petter Langtangen

University of Oslo, Moltke Moes vei 35, 0851 Oslo, Norway
Center for Biomedical Computing at Simula Research Laboratory, P.O. Box 134, N-1325 Lysaker, Norway

ARTICLE INFO

Article history:
Received 6 October 2015
Received in revised form
5 February 2016
Accepted 9 February 2016
Available online 26 February 2016

Keywords:
CFD
Python
Cython
DNS
Slab
Pencil
FFT
MPI

ABSTRACT

Direct Numerical Simulations (DNS) of the Navier Stokes equations is an invaluable research tool in fluid dynamics. Still, there are few publicly available research codes and, due to the heavy number crunching implied, available codes are usually written in low-level languages such as C/C++ or Fortran. In this paper we describe a pure scientific Python pseudo-spectral DNS code that nearly matches the performance of C++ for thousands of processors and billions of unknowns. We also describe a version optimized through Cython, that is found to match the speed of C++. The solvers are written from scratch in Python, both the mesh, the MPI domain decomposition, and the temporal integrators. The solvers have been verified and benchmarked on the Shaheen supercomputer at the KAUST supercomputing laboratory, and we are able to show very good scaling up to several thousand cores.

A very important part of the implementation is the mesh decomposition (we implement both slab and pencil decompositions) and 3D parallel Fast Fourier Transforms (FFT). The mesh decomposition and FFT routines have been implemented in Python using serial FFT routines (either NumPy, pyFFTW or any other serial FFT module), NumPy array manipulations and with MPI communications handled by MPI for Python (`mpi4py`). We show how we are able to execute a 3D parallel FFT in Python for a slab mesh decomposition using 4 lines of compact Python code, for which the parallel performance on Shaheen is found to be slightly better than similar routines provided through the FFTW library. For a pencil mesh decomposition 7 lines of code is required to execute a transform.

1. 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 order and accuracy 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, see, e.g., [1,2]. Normally, this demands a need for developing tailored, hand-tuned codes in what we here call low-level languages: Fortran, C or C++ (despite the possibility for creating high-level abstractions in Fortran 90/2003 and C++, the extreme performance demands of DNS codes naturally leads to minimalistic use of classes and modules). Few DNS codes are openly available and easily accessible to the public and the common fluid mechanics researcher. Some exceptions are hit-3d (Fortran90) [3], Philofluid (Fortran) [4], Tarang (C++) [5], and Turbo (Fortran90) [6]. However, the user interfaces to these codes are not sophisticated and user-friendly, and it is both challenging and time consuming for a user to modify or extend the codes to satisfy their own needs. This is usually the nature of codes written in low-level languages.

* Corresponding author at: University of Oslo, Moltke Moes vei 35, 0851 Oslo, Norway.
E-mail address: mikaem@math.uio.no (M. Mortensen).

http://dx.doi.org/10.1016/j.cpc.2016.02.005
0010-4655/

It is a clear trend in computational sciences over the last two decades that researchers tend to move from low-level to high-level languages like Matlab, Python, R, and IDL, where prototype solvers can be developed at greater comfort. 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.

Python is a high-level 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.

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 scientific Python modules like NumPy (cf. [7,8]), used for linear algebra and array manipulations, and MPI for Python (mpi4py) [9], 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 requires reimplementations in Fortran, C or C++. In this paper, we conquer this misconception: The only real requirement for developing a fast scientific Python solver is that all array manipulations are performed using vectorization (that calls underlying BLAS or LAPACK backends or compiled NumPy ufuncs) such that explicit for loops over long arrays in Python are avoided. The MPI for Python module in turn provides a message passing interface for NumPy arrays at communication speeds very close to pure C code.

There are already several examples on successful use of Python for high-performance parallel scientific computing. The sophisticated finite element framework FEniCS [10] 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 is spent within the low-level wrapped C++ functions that perform the costly linear algebra operations [11]. 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 [13] is a major provider of linear algebra to the open source community. PETSc was developed in C, but through the package PETSc for Python (petsc4py) almost all routines may be set up and called from Python. PyClaw [14] is another good example, providing a compact, powerful, and intuitive Python interface to the algorithms within the Fortran codes Clawpack and SharpClaw. PyClaw is parallelized through PETSc and has been shown to scale well up to 65,000 cores.

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. As such, 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 on thousands of processors. To establish scaling and benchmark results, we have run the codes on Shaheen, a massively parallel Blue Gene/P machine at the KAUST Supercomputing Laboratory. The code described is part of a larger DNS project and available online (https://github.com/mikaem/spectralDNS) under a GPL license.

2. The Navier–Stokes equations in spectral space

Our DNS implementation is based on a pseudo-spectral Fourier–Galerkin method [15] 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 uniformly in space using a structured computational mesh with N points in each direction, the mesh points can be represented as[1]

$$\boldsymbol{x} = (x, y, z) = \left\{ (x_i, y_j, z_k) = \left(\frac{2\pi i}{N}, \frac{2\pi j}{N}, \frac{2\pi k}{N} \right) : i, j, k \in 0, \ldots, N-1 \right\}. \tag{5}$$

In the spectral Galerkin method all variables must be transformed from the physical mesh $\boldsymbol{x}$ to a discrete and bounded Fourier wavenumber mesh. The three-dimensional wavenumber mesh may be represented as

$$\boldsymbol{k} = (k_x, k_y, k_z) = \left\{ (l, m, n) : l, m, n \in -\frac{N}{2} + 1, \ldots, \frac{N}{2} \right\}. \tag{6}$$

The discrete Fourier transforms are used to move between real space $\boldsymbol{x}$ and spectral space $\boldsymbol{k}$. A component of the velocity vector (with similar notation for other field variables) is approximated in both real and spectral spaces as

$$u(\boldsymbol{x}, t) = \frac{1}{N^3} \sum_{\boldsymbol{k}} \hat{u}_{\boldsymbol{k}}(t) e^{\imath \boldsymbol{k} \cdot \boldsymbol{x}}, \tag{7}$$

$$\hat{u}_{\boldsymbol{k}}(t) = \sum_{\boldsymbol{x}} u(\boldsymbol{x}, t) e^{-\imath \boldsymbol{k} \cdot \boldsymbol{x}}, \tag{8}$$

where $\hat{u}_{\boldsymbol{k}}(t)$ is used to represent the Fourier coefficients, $\imath = \sqrt{-1}$ represents the imaginary unit, and $e^{\imath \boldsymbol{k} \cdot \boldsymbol{x}}$ represents the basis functions for the spectral Galerkin method. Eqs. (7) and (8) correspond, respectively, to the three-dimensional discrete Fourier

[1] Different domains lengths and number of points in each direction are trivially implemented, and we use a uniform mesh here for simplicity.