Journal of Fundamental and Applied Research **Vol. 1**, Issue 1 (2021) 20210004

Received: 13 December 2021 / Accepted: 18 December 2021
Published online: 22 December 2021
© The Author(s) 2021

Speeding up Numerical Python Programs

Askar B. Abdikamalov[1, 2, 3, *]

[1] *Center for Field Theory and Particle Physics and Department of Physics, Fudan University, 200438 Shanghai, China*
[2] *Ulugh Beg Astronomical Institute, Astronomy Str. 33, Tashkent 100052, Uzbekistan*
[3] *Institute of Fundamental and Applied Research, National Research University TIIAME, Kori Niyoziy 39, Tashkent 100000, Uzbekistan*

Python general-purpose, highly-level programming language that is currently used widely in scientific computing. It has several attractive features such as ease of code readability, while its syntax allows to express concepts in fewer lines of code than would be possible in traditional languages such as C. The main drawback of Python is its relatively slow speed in executing numerical operations. Several methods to accelerate such computations. In this work, we evaluate how such methods perform for a numerical solution of a partial differential equation.

I. INTRODUCTION

Development of software for scientific computations often takes large fraction of the total effort in computational science. Using development tools that save code development time and effort is thus of tremendous importance. Recently, Python attracted enormous attention as such language for several reasons. First, Python is a free, open-source, well-designed general-purpose, high-level programming language. The philosophy behind its design focuses on code readability, and its syntax allows to express concepts in fewer lines of code than would be possible in traditional languages such as C. The language provides constructs aimed to enable clear programs on both a small and large scale. Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. It features a dynamic type system and automatic memory management and has a large and comprehensive standard library. Like other dynamic languages, Python is often used as a scripting language, but is also used in a wide range of non-scripting contexts. Finally, most of the features which have made MATLAB so widely popular are also available in Python.

The most serious drawback of Python when using for scientific computations is that there is a possible loss of computational speed. Numerical algorithms often use loops over array data structures. In standard Python, loops overs long lists or arrays run slowly. The speed can still be acceptable to many problems such as solution one-dimensional ordinary partial differential equations, but for large-scale problems the slow speed becomes unacceptable. Several methods and software have been developed that addresses this problem. The examples are numpy [1], vectorization, Cython [2], PyPy (`pypy.org`). In this paper, we investigate how much speed up each of these techniques offer for typical scientific computations. To this end, we solve a standard wave equation in a heterogenous medium in two space dimensions.

II. PYTHON ACCELERATORS

There are several ways of improving the speed of slow loops over large arrays in Python. First, one can replace the loop by a set of operations on entire arrays. This is referred to as vectorization. NumPy is an extension to the Python programming language, adding support for large, multidimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays (`numpy.org`). Because Python is currently implemented as an interpreter, mathematical algorithms written in it often run slower than compiled equivalents. Numpy seeks

*Electronic address: aaskar17@fudan.edu.cn

to address this problem for numerical algorithms by providing multidimensional arrays and functions and operators that operate efficiently on arrays. Thus any algorithm that can be expressed primarily as operations on arrays and matrices can run almost as quickly as the equivalent C code.

Another approach, one can move the loops to a code that can be compiled. Such a code can either be in Fortran, C, or C++. There are a wide range of tools and programs for simplifying development of such codes. Examples are F2PY [3], SWIG [4], etc. Cython [5] is a newly developed approach that enhances the Python language with new syntax that enables automatic compilation of code pieces to machine code and thus achieving significant code speed-up. One can also manually write all the necessary C code for migrating loops to C, without relying on any tool for automating the (quite complicated) communication of data between C and Python.

PyPy is a Python interpreter and just-in-time compiler. PyPy focuses on speed, efficiency and compatibility with the original CPython interpreter [6]. PyPy was conceived as an implementation of Python written in a programming language that is similar to Python. This makes it easy to identify areas where it can be improved and makes PyPy more flexible and easier to experiment with than CPython. PyPy aims to provide a common translation and support framework for producing implementations of dynamic languages, emphasizing a clean separation between language specification and implementation aspects. It also aims to provide a compliant, flexible and fast implementation of the Python programming language using the above framework to enable new advanced features without having to encode low level details into it.

III. THE BENCHMARK PROBLEM

In our benchmark problem, we employ a standard wave equation in two space dimensions. The description of the problem closely follows excellent presentation in [2].

We investigate the performance of different programming tools and techniques on a not too complicated mathematical problem, that has a real application in mechanics and that contains algorithmic widespread important steps in more advanced computational mechanics applications. Our choice for benchmark is then a standard wave equation in a heterogeneous medium with local wave velocity k:

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot [k \nabla u] \tag{1}$$

For our tests, the attention is restricted to two space dimensions:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(k(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(k(x,y)\frac{\partial u}{\partial y}\right) \tag{2}$$

The boundary condition are set to $u = 0$ for the whole boundary of a rectangular domain $\Omega = (0,1) \times (0,1)$. Further, the initial value of u is $I(x,y)$ at $t = 0$ while $\frac{\partial u}{\partial t} = 0$. The wave initial shape is chosen as a Gaussian bell function,

$$I(x,y) = A \exp\left(\left(\frac{x - x_c}{2\sigma_x}\right)^2 + \left(\frac{y - y_c}{2\sigma_y}\right)^2\right) \tag{3}$$

with $A = 2, x_c = y_c = 0.5$, and $\sigma_x = \sigma_y = 0.15$. The defination of the function k is $k(x,y) = \max(x,y)$, which causes the waves to move along the line between the corners $(0,0)$ and $(1,1)$ of the domain instead of uniformly over the grid.

The following finite difference scheme is used to solve the wave equation:

$$\begin{aligned} u^l_{i,j} = &\left(\frac{\Delta t}{\Delta x}\right)^2 \left[k_{i+\frac{1}{2},j}\left(u_{i+1,j} - u_{i,j}\right) - k_{i-\frac{1}{2},j}\left(u_{i,j} - u_{i-1,j}\right)\right]^{l-1} + \\ &+ \left(\frac{\Delta t}{\Delta y}\right)^2 \left[k_{i,j+\frac{1}{2}}\left(u_{i,j+1} - u_{i,j}\right) - k_{i,j-\frac{1}{2}}\left(u_{i,j} - u_{i,j-1}\right)\right]^{l-1} \end{aligned} \tag{4}$$

Here, $u^l_{i,j}$ denotes u at the grid point x_i and y_j at time level t_l, where

$$\begin{gathered} x_i = i\Delta x, i = 0, \ldots, n, \\ y_j = j\Delta y, j = 0, \ldots, m \text{ and} \\ t_l = l\Delta t. \end{gathered}$$

Copyright © 2021

Also, $k_{i+\frac{1}{2},j}$ represents for $k\left(x_{i+\frac{1}{2}}, y_j\right)$.

Note, the finite difference scheme is explicit and only conditionally stable. The largest possible time step is chosen as,

$$\Delta t = \frac{1}{\max_{xy} k(x,y)} \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-\frac{1}{2}}.$$

The benchmark relevance problem extends far beyond physical problems that can be modeled by the standard wave equation. In essence, the resulting computer code will be written to perform an update of a variable-coefficient Laplace operator, i.e., an operation of the form

$$u_{\text{new}} = Q + \nabla \cdot [k \nabla u_{\text{old}}]$$

where Q denotes some known quantity. Such updates arise in a wide range of physical problems and numerical methods. Therefore, the examined code segment in this paper will be relevant to most finite difference-based software for fluid flow, heat transport, and electrostatics. For example, an implicit scheme for a heat transport problem with nonlinear heat conduction can be considered. If convection and diffusion are splitted in two steps, the diffusion step will lead to a linear system that must be solved by iterative methods. If a Conjugate Gradient method is used for this purpose, combined with a Multigrid preconditioner, typically involves the discrete differential operator in the smoother step of the Multigrid algorithm. This smoother step will act as an update from u_{old} to u_{new} of the type shown above, and k can be computed from u_{old} values. Likewise, finite difference schemes for the Navier-Stokes equations apply a splitting of the equations, with the same kind of Laplace operator steps as in our wave equation. The benchmark problem main limitation is the use of structured grids and finite difference operators.

The scheme then leads to a linear system we have an implicit scheme for, say a heat transport problem, an iterative numerical solution strategy, say a multigrid method, will result in updates of the type we have for an explicit scheme for the wave equation.

In the scheme, we need k values between the grid points, as in $k_{i+\frac{1}{2},j}$. This is a straightforward computation if we know k as an explicit mathematical function, which is the case in the present problem. However, in the more general case, $k(x, y)$ is obtained from measurements and will normally be known only over a grid. If we sssume that k is only available at the same grid as we use for u, we have to approximate quantities like $k_{i+\frac{1}{2},j}$ by k values at the grid points. The most straitforward approximation is the arithmetic mean,

$$k_{i+\frac{1}{2},j} \approx \frac{1}{2}\left(k_{i,j} + k_{i+1,j}\right) \tag{5}$$

with similar definitions for $k_{i-\frac{1}{2},j}, k_{i,j+\frac{1}{2}}$, and $k_{i,j-\frac{1}{2}}$. For problems that involve highly discontinuous media (e.g., waves in geological media), it is more natural to apply the harmonic mean:

$$k_{i+\frac{1}{2},j} \approx \left(\frac{1}{2}\left(\frac{1}{k_{i,j}} + \frac{1}{k_{i+1,j}}\right)\right)^{-1} \tag{6}$$

In the present problem, it turns out that whether an arithmetic or harmonic mean is applied has substantial effect on the relative performance of different programming tools and techniques. The reason is that the harmonic mean results in a significantly more work, which may then dominate over the array look-ups in the scheme (for the fastest implementation the harmonic mean computation consumes about 80% of the time of an update in the scheme). We may consider the arithmetic and harmonic mean alternatives as two (physically motivated) models for two classes of the complexity of a diffusion operator in more complicated partial differential equations. The arithmetic mean denotes a model where the k coefficient is quickly evaluated, while the harmonic mean leads to a model with a significant set of arithmetic operations, which typically comes out in nonlinear diffusion operators, perhaps with complicated constitutive laws. The latter case may lead to calling mathematical functions (e.g., the exponential function) as part of the $k_{i+\frac{1}{2}}$ computation, which results in even more work than for the harmonic mean. It therefore preferable to test mean as models for "simple" variable-coefficient Laplace operator.

Our algorithm nwo can be summarized as follows:

Copyright © 2021

Define $u_{i,j}, up_{i,j},$ and $um_{i,j}$ to represent $u^{l}_{i,j}, u^{l+1}_{i,j}$ and $u^{l-1}_{i,j}$
SET THE INITIAL CONDITIONS:
$u_{i,j} = I(x, y),$ for $i = 0, \ldots, n$ and $j = 0, \ldots, m$
CALCULATE THE FIRST TIME STEP
$t = 0$
START THE TIME LOOP:
while time $t \leq t_{\text{stop}}$
$t \leftarrow t + \Delta t$
SOLVE FOR CURRENT TIME STEP:
for $i = 1, \ldots, n-1$:
for $j = 1, \ldots, m-1$:
calculate $up_{i,j}$ from (3)
SET BOUNDARY CONDITIONS
INITIALIZE FOR NEXT STEP:
$um = u$
$u = up$

IV. SET OF TESTS

For all programs, the time spent by the main time loop is considered. The time spent for setting the initial condition and calculating the first time step, which needs to be treated separately, is ignored. Compared to the execution time of all the other time steps, this time is usually negligible.

The allocation and initialization of data structures is in pure Python for all implementations as well as the time loop. Only the space loop that contains the variable-coefficient Laplace operator can be a subject to various optimization techniques. We have also tested timings where the time loop has been migrated to compiled code, but the gain in speed was negligible, which reminds us that the need for optimization is usually limited to small, very computing intensive parts of a numerical code.

There are two ways of evaluating the variable coefficient $k_{i,j}$ in the scheme: either k can be implemented as a Python function, or k values are stored at the spatial grid points in an array. The latter approach has been adopted, since it is much more efficient than calling a function.

All the timings reported in Table 2 are done for three different types of numerical grid: medium resolution is a 250×250 grid with 707 time steps using the arithmetic mean for k. For high and low resolutions all the parameters of the medium numerical grid are increased and decreased by 50% respectively.

The timings have been done on a Linux laptop with the setup as shown in Table 1. The laptop is a Dell Inspiron N4010 with a 2.53GHz processor and 2GB of memory running Ubuntu 12.10.

V. DISCUSSION AND CONCLUSION

Table 1: Machine Configuration

Software	Version
Python	2.7.1
Cython	0.18.1
NumPy	1.7.0
PyPy	1.9

Table 2 : Timing results

Copyright © 2021

	Low resolution	Medium resolution	High resolution
Pure Python with u[i][j]	13.17276406	103.727752	354.7250731
Using numpy and u[i,j]	78.08817911	654.2926378	2118.943949
Using numpy with vectorization and u[i,j]	0.473357916	4.88210296	21.40271497
Compiled code with c type definations	0.125884056	0.969850779	3.712311983
Compiled code with numpy and u[i,j]	75.14957094	585.8985472	2114.823879
Compiled code with numpy and vectorization and u[i,j]	0.510840178	4.862480164	23.30624304
Pure Python using pypy-1.9	4.366171122	4.471300125	4.586941004

In section we briefly report our main findings about the efficiency of different speed-up tools and implementation constructs. Table 2 shows the amount of time speed spent for computation with each of the methods for each of the three resolutions. As we can see, pure Python or Python with numpy yield slowest results. Numpy with vectorization leads to a dramatic speed of about to orders of magnitide. Here, medium resolution is about 10 times slower than the low resolution. High resolution run is about 4 times slower than the low resolution run. Compiled see code is even faster roughly by a factor of several. Compiled versions of the last two yield comparable results. Finally, pure Python using PyPy of version 1.9 yield about 4.5, largerly insensetive to the grid resolution.

Summarizing, in this paper we have demonstrated that computational Python accelerators can yield dramatic speed up for typical scientific computations. This is particularly true for numpy with vectorization, compiled C loops, and PyPy. Therefore, we highly recommend their usage for scientific computations.

acknowledgements

The research is supported in part by Grants F-FA-2021-432, F-FA-2021-510, and MRB-2021-527 of the Uzbekistan Ministry for Innovative Development.

[1] H. P. Langtangen. Python Scripting for Computational Science. Springer-Verlag Berlin Heidelberg, 2004.
[2] I. Wilbers, H. P. Langtangen, and Å. Ødegård. Using Cython to Speed up Numerical Python Programs In: Proceedings of MekIT'09, ed. by B. Skallerud and H. I. Andersson, pp. 495-512, NTNU, Tapir (ISBN: 978-82-519-2421-4).
[3] P. Peterson, F2PY: Fortran to Python interface generator http://scipy.org/F2py.
[4] D. Beazley et al. SWIG: Simplified wrapper and interface generator http://www.swig.org.
[5] G. Ewing, R. Bradshaw, S. Behnel, D. S. Seljebotn et al. Cython: C-extensions for Python. http://cython.org.
[6] S. Pedroni, 2007. "PyPy - Goals and Architecture Overview".
100

Copyright © 2021