Performance Comparisons for Python Libraries in Parallel Computing and Physical Simulation

Mr. Olubunmi Gregory Adekanmbi, Prairie View A&M University

Performance Comparisons for Python Libraries in Parallel Computing and Physical Simulation

Olubunmi Adekanmbi, Lei Huang

Computer Science Department
Prairie View A&M University

Abstract

Physical simulation today requires fast and efficient parallel computing to achieve the accuracy and performance. At the core of physical simulation lies the tools and frameworks that are driving it, from programming language, compilation, algorithms and high-performance computing. Python is a high-level programming language of choice favored by many developers and researchers since it is very productive. However, Python is also notorious for its poor performance due to the interpreted mode and its internal global interpreter lock (GIL). In this paper, we overcome the performance problem inherited in Python by using three different Python computing libraries. The work demonstrates that by using the right computing library, Python may achieve both high productivity and high performance in physical simulations.

Keywords

Physical simulation, Python, Parallel computing, Performance.

1. Introduction

Python as a high-level programming language, it has become the most popular programming language since 2018 according to the PYPL Index. Python has been widely used in data science, machine learning, Web development, and other software development due to its high productivity. However, Python has not been widely accepted by the scientific computing community, especially for large scale scientific simulation and modeling. The main reason is that Python does not provide high performance to allow researchers to fully utilize the sophisticated resources in supercomputers. In this paper, we demonstrate that it is feasible to achieve high performance in physical simulations using a simple case.

There are different Python parallel libraries that are available today with the main aim of ensuring Python codes run faster in parallel to break the GIL, which is essential to promote Python as a high-performance programming language. With the new Python parallel libraries, physical simulations can be executed successfully on GPUs and multicores. Taichi, NumPy and Numba are Python libraries designed for high-performance numerical computing and machine learning. In this paper we introduce these Python libraries / frameworks and use them to implement several physical simulations. We evaluate the performance of these libraries and discuss the advantages and disadvantages in physical simulations. We will also discuss how to apply them in both simulations and machine learning applications.

To accomplish the research, we chose Taichi, NumPy and Numba to start with because they were specifically designed for high performance computing. For us to thoroughly compare these libraries we must first define several physical simulations, understand the physics behind the simulations and implement them using these libraries i.e., have the same simulation written with Taichi, NumPy and Numba. One of the simulations we are implementing is the N-body problem. We will then go further by comparing the length of the overall codes and how long it took to execute them individually. We will repeat this for several simulations as well, once that is completed, we are then able to document and compare the results. The work will provide an informed decision on which of the libraries to adopt for both simulations and machine learning applications. We believe with these comparisons and having identified the advantages and disadvantages; we can proceed to creating functional programs/instructions.

Finally these three libraries benefit from extensive documentation, technical support, a great community of contributors and various built-in assets. Ultimately, the choice of library depends on various factors like: speed, compliance with problems, complexity, readability and future support. At the end of this paper we will summarize our findings and share a conclusion based on our experiment.

Our paper is motivated by the growing interest among scientists and researchers across the globe. In today's world, the need for fast and efficient parallel computing tools and frameworks for physical simulation has become a topic of interest. We discovered the need to have a high performance framework to simulate physical problems and soft materials; Python as a whole wasn't doing justice to that, hence the need to select a library that can accommodate the need and this led us to the experiment of comparing different Python libraries to simulate the N-Body problem

2. Background

Many physical processes can be modeled using a particle system in which each particle interacts with all other particles according to physics principles. From astronomical simulations of celestial motions to electrostatic interactions between molecules.

The N-body challenge is the difficulty of predicting the motion of a set of N objects that interact with one another independently over a long range (usually gravitationally or electrostatically). Formally, for a group of N objects in space, if the initial positions (x_0) and velocities (v_0) are known at time t_0, predict the positions (x) and velocities (v) of the N objects at a later time t. Solving this problem was originally motivated by the need to understand the motion of the Sun, planets, and the visible stars, but it has been applied to galaxies, planets, fluids, and molecules.

Below is a brief description of the three Python computing libraries of focus:

2.1 Taichi

Taichi is a high-performance programming language embedded in Python for computer graphics applications. The design goals are:

- **Productivity and portability**: easy to learn, to write, and to share
- **Performance**: data-oriented, parallel, mega-kernels
- **Spatially sparse programming**: save computation and storage on empty regions
- **Decouple** data structures from computation
- **Differentiable** programming support

Taichi is different from other libraries like TensorFlow, PyTorch, NumPy, JAX etc. because it uniquely supports mega-kernels and spatial sparsity. This framework supports Windows, Linux, and OS X. and runs on both CPUs and GPUs (CUDA/OpenGL/Apple Metal)

2.2 Numba

Numba is a just-in-time compiler for Python that works best on code that uses NumPy arrays and functions, and loops. The most common way to use Numba is through its collection of decorators that can be applied to your functions to instruct Numba to compile them. When a call is made to a Numba-decorated function it is compiled to machine code "just-in-time" for execution and all or part of your code can subsequently run at native machine code speed!

Numba works perfectly with OS: Windows, OSX, Linux. Also supports M1/Arm64, GPUs: Nvidia CUDA.

2.3 NumPy

NumPy as the name implies, Numerical Python is a fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

NumPy is open-source and the most important object defined in NumPy is an N-dimensional array type called **ndarray**.

3. Methodology

To compare the performance of NumPy, Numba and Taichi Python Libraries, we first implement the

N-Body simulation separately using these three libraries. We need to understand the physics of the simulation, mathematical formula, and the features of these three Python libraries. Our overall experience with each Python library was comparable, though it is important to note that each library has its specific strength which will further be discussed.

Below are the basics of the Gravitational N-Body Problem for the sake of this experiment. Finding positions and movements of bodies in space subject to gravitational forces from other bodies using Newton's laws of motion.

Gravitational force F between two bodies of masses m_a and m_b as seen in **Figure 1** is:

$$F_{ab} = \frac{Gm_a m_b}{r^2} \tag{1}$$

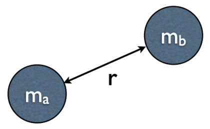

Figure 1: Gravitational N-Body Problem

G is the gravitational constant (6.673×10^{-11} $m^3 kg^{-1} s^{-2}$) and r the distance between the bodies.

$$r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2} \tag{2}$$

For a system of N particles, the sum of the forces is:

$$F = \sum_{i<j} F_{ij} = \sum_{i<j} \frac{Gm_i m_j}{r_{ij}^2} \tag{3}$$

3.1 Simple N-Body Scenario with N=4

Thanks to Newton's Third Law ("for every action, there is an equal and opposite reaction").

However, this is still a complicated problem that only grows in complexity with N.

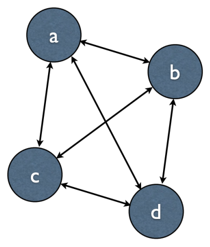

Figure 2: Simple N-Body Scenario with N=4

Subject to forces, a body accelerates according to Newton's second law: F = ma where m is mass of the body, F is force it experiences and a is the acceleration.

Let the time interval be Δt. Let vt be the velocity at time t. For a body of mass m the force is:

$$F = m\frac{v_{t+1} - v_t}{\Delta t} \tag{4}$$

New velocity then becomes

$$v_{t+1} = v_t + \frac{F \cdot \Delta t}{m} \tag{5}$$

Over time interval Δt position changes by:

$$x_{t+1} = x_t + v.\Delta t \tag{6}$$

It is important to note that once the bodies move to new positions as seen from the arrows in **Figure 2**, forces change and computation has to be repeated.

After establishing how this works mathematically, we then went further to compute them using Python and leveraging the libraries earlier discussed.

3.2 Using Taichi

One distinguishing feature of Taichi is that it provides a new language extension to Python and uses a Just-In-Time (JIT) compiler to optimize the Python code and generate code for specific devices, such as NVIDIA GPUs and APPLE Meta[1]. A developer needs to define the memory layout of the computing data using the keyword Fields, which is borrowed from Physics. Once the data layout is defined, the JIT is able to optimize the data access for the specific devices. It allows developers to define kernels, which are built and run on specific devices.

To achieve good performance in this Taichi version of the N-Body simulation, we leverage its just-in-time compiler to offload compute-intensive tasks to multi-core CPUs and massively parallel GPUs[2]. We use the keyword @ti.kernel to define the force calculation and position update functions, which offload both of them on the accelerator device. The entire memory required in the simulation is predefined to allow compiler optimization. The output of the simulation is shown in **Figure 3** below and the source code can be accessed in our Github repository.

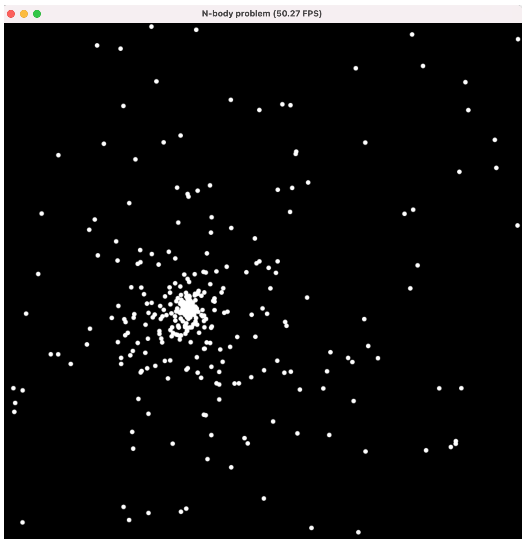

Figure 3: Taichi N-Body simulation

3.3 Using Numba

Numba is an open source JIT compiler that translates Python code into fast machine code. It has built-in automatic parallelism capability if decorators are giving. The simplest way to parallelize a Python code using Numba is to use the decorator @jit(nopython=True, parallel=True)[5]. It can generate Numpy Universal Functions(ufuncs) that supports in nopython mode, and generates vectorized or multi-threaded code. It may also parallelize an explicit parallel loop once the parallel=True is given. The JIT compiler may detect a reduction pattern to further optimize the parallel code. However, if the code has dependencies, Numba is not able to solve them and has to execute it in sequence.

3.4 Using NumPy

Numpy is the most widely used computing package in Python to deal with multi-dimensional arrays. It was written in C with a wrapper of the Python interface, which makes the code very efficient. Numpy depends on an accelerated linear algebra library such as Intel MKL or OpenBLAS for high-performance linear algebra computing. It also provides Universal functions (ufunc) which allow an operation to be performed element by element on whole arrays and execute the operations in parallel[4].

4. Evaluation

For the sake of evaluation and accurate comparison, we used the timeit() method to measure the execution time for each function within the program. **Table 1** below shows how each framework performed on a Quad-Core CPU with respect to the functions within it.

Functions	Taichi (seconds)	NumPy (seconds)	Numba (seconds)
Initialize	0.7882241	0.0114017	0.7692649
Compute Force	0.04270517	2.48655686	0.70108952
Update	0.02413331	0.00542014	0.35490218
TOTAL	**0.85506258**	**2.5033787**	**1.8252566**

Table1: Performance numbers per each function in each loop step.

The initialization function takes more time in Taichi and Numba since it requires a JIT compiler to compile the code at runtime. The initialization is only running once during the program so it is not important. The critical functions are computer_force() and Update(), the former is to compute the force applied to each object based on the distances among all other objects, and the update() function updates the velocity and position accordingly. These two functions are executed many times during the simulation so that their performance matters. As shown in **Table 1**, Taichi performs the best for these two functions since they are running on a GPU, Numba has 10X worse performance in both of them. Numpy is the worst since the code cannot utilize the ufunc feature provided by Numpy.

5. Conclusions

We have been able to describe and implement three(3) different programs for the N-Body simulation, and we have described the time taken for these programs to successfully execute. One major insight we derived and also from the table above is that Taichi performs best with the function that requires major numerical calculations. Therefore, if an algorithm depends heavily on numerical calculations then it is likely Taichi will provide significant speedups. Also, Taichi was able to prove it's strength in computing force and preparation for execution to GUI. After execution, the N-bodies in the Taichi program moved faster within the simulation.

We hope we have brought some clarity in the trade-off between these three libraries which will provide information to engineers and scientists about which framework is best fit for them. You may access the source code of the paper in Github: https://github.com/kombolee/N-Body_Simulation

References

1. Yuanming Hu: The Taichi Programming Language – A hands-on tutorial
   https://yuanming.taichi.graphics/publication/2020-taichi-tutorial/taichi-tutorial.pdf

2. (SIGGRAPH Asia 2019) Taichi: High-Performance Computation on Sparse Data Structures
   EditSign URL: https://yuanming.taichi.graphics/publication/2019-taichi/taichi-lang.pdf

3. (ICLR 2020) DiffTaichi: Differentiable Programming for Physical Simulation URL:
   https://arxiv.org/abs/1910.00935

4. Travis E. Oliphant, August 2018: Guide to NumPy Vol 1. Trelgol Publishing

5. Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A llvm-based python jit compiler. In Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (pp. 1–6).

6. Samuel S. Cho: N-Body Problem – Lecture 12, Wake Forest University, Fall 2011 - CSC 391/691: GPU Programming (pp. 1-6) - https://users.wfu.edu/choss/CUDA/docs/Lecture%2012.pdf

Olubunmi Adekanmbi
Olubunmi Adekanmbi is a Graduate Student in Computer Science at Prairie View A&M University. His interests are in Artificial Intelligence and Cloud Computing.

Lei Huang
Dr. Lei Huang serves as an Associate Professor in the Computer Science Department of Prairie View A&M University. His research interests are in High Performance and Cloud Computing, Data Science and Computational Science.