A Comparative Evaluation of Parallel Programming Python Tools for Particle-in-Cell on Symmetric Multiprocessors

Oscar Blandino H.[1(✉)] and Esteban Meneses[1,2]

[1] School of Computing, Costa Rican Institute of Technology, Cartago, Costa Rica
oscar.blandino.hernandez@intel.com, emeneses@cenat.ac.cr
[2] Advanced Computing Laboratory, National High Technology Center, San Jose, Costa Rica

Abstract. The Python programming language has established itself as a popular alternative for implementing scientific computing workflows. Its massive adoption across a wide spectrum of disciplines has created a strong community that develops tools for solving complex problems in science and engineering. In particular, there are several parallel programming libraries for Python codes that target multicore processors. We aim at comparing the performance and scalability of a subset of three popular libraries (Multiprocessing, PyMP, and Torcpy). We use the Particle-in-cell (PIC) method as a benchmark. This method is an attractive option for understanding physical phenomena, specially in plasma physics. A pre-existing PIC code implementation was modified to integrate Multiprocessing, PyMP, and Torcpy. The three tools were tested on a manycore and on a multicore processor by running different problem sizes. The results obtained consistently indicate that PyMP has the best performance, Multiprocessing showed a similar behavior but with longer execution times, and Torcpy did not properly scale when increasing the number of workers. Finally, a just-in-time (JIT) alternative was studied by using Numba, showing execution time reductions of up to 43%.

Keywords: Parallel programming · Python · Particle-in-cell

1 Introduction

Particle-in-cell (PIC) is one of the most important computational methods in physics to study problems in solid, fluid mechanics, but specially in plasma. It solves a set of partial differential equations with a combination of individual particles on a Lagrangian frame and moments computed on Eulerian mesh points. The first PIC simulations were performed in late 1950 s using between 100 and 1,000 particles to simulate the motion and interaction between them. Nowadays, this kind of simulations are performed using between 10^5 and 10^{10} particles,

P. Navaux et al. (Eds.): CARLA 2022, CCIS 1660, pp. 1–15, 2022.
https://doi.org/10.1007/978-3-031-23821-5_1

representing a challenge for computer systems. Large PIC simulations require the use of supercomputers and code optimizations to reduce execution time [5].

Traditional programming languages in HPC, such as FORTRAN and C/C++, were used to implement PIC and other well-established numerical methods. However, the community working on computational science adopted Python as a popular option for running simulations. A fundamental reason for that change is the evolution of problem-solving approaches [15]. Python is easier to learn and use than FORTRAN and C/C++. But, it still has great tools for scientific computing (SciPy, NumPy, Matplotlib, PyTorch). The first scientific computing research projects were based on mathematical models simpler than the complex models used by scientists today. The knowledge the scientific community gained over the previous decades led to the development of more complex models to understand bigger or more difficult problems at a higher precision. In fact, it is now common to include machine learning methods within the workflow of scientific applications. The high popularity of Python across scientific disciplines, the broad availability of tools, and a huge user base, has made Python an attractive option for implementing complex mathematical models and simulations.

Chips with multiple cores dominate the processor market these days. The architecture trend is to increase the number of cores per processor. As Moore's Law still holds true, we can only expect this course of action to persist for a few more years. The latest release of the Top500 list [17] shows that nearly 70% of the systems solely rely on symmetric multiprocessors (SMP) for their computing power (no accelerators). It is therefore crucial to address the performance characteristics of parallel programming Python tools for SMP architectures when implementing PIC methods.

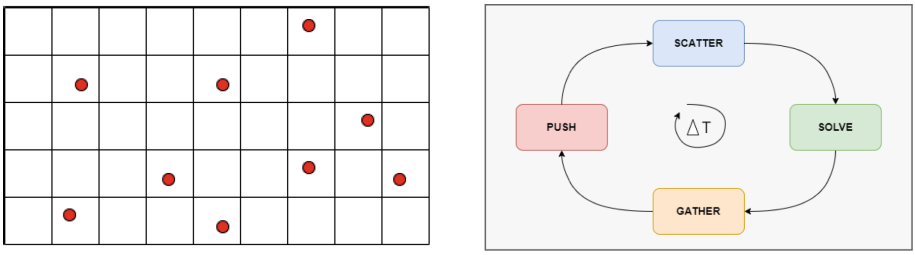
This paper sets out to explore three popular parallel programming Python tools for SMP architectures. We use the PIC method as a guiding example. To the best of our knowledge, this is the first study on that topic. Our contribution is twofold. First, we provide a picture of the features these tools provide when implementing a PIC method. Second, we present a comparative analysis of those tools backed up with experimental results on two different SMP architectures.

2 Background

2.1 Particle-in-Cell

Mathematical Base. Particle-in-cell (PIC) is a method used to model physical systems whose behavior varies at macro and micro levels. At the macro level, the electromagnetic fields are calculated using Maxwell's equations. At the micro level, the position, velocity, charge, and current density properties are calculated for a set of particles [5,13]. The main objective of the PIC method is to simulate the motion of plasma particles based on the interaction of position and velocity of the particles, with self induced and external electromagnetic fields. To simulate this dynamic, the PIC model uses a grid, as presented in Fig. 1a. In that grid, the position of each particle is shown. The grid is used to calculate and

determine the interaction of the particle with electromagnetic fields, and subsequently the particle's new position and velocity. These particles, depending on the application, could have more assigned properties, such as mass, charge, and material. The particles are the ones responsible of transporting mass and energy through the grid [13].



(a) Particle-in-cell Grid. (b) Particle-in-cell Phases.

Fig. 1. Particle-in-cell (PIC) method.

Particle-in-cell simulations normally solve the equations of motion of N particles with the Newton-Lorentz's force [5,9], considering the non-relativistic case, and also solve Maxwell's equations to calculate electromagnetic fields, charge and current density. Considering N particles, with $i = 1, \cdots, N$, the motion equations are presented in Eqs. 1 and 4 of Fig. 2. Variables X_i and V_i correspond to the position and velocity of particle i. Also, e_i and m_i correspond to the electric charge and mass of particle i. Finally, E and B correspond to the electric and magnetic fields. On the other hand, Maxwell's equations are presented in Eqs. 2, 3, 5, and 6. Variable ε corresponds to the permittivity of the medium, H is the magnetic field, J corresponds to the current density, and ρ to the charge density.

$$\frac{dX_i}{dt} = V_i \quad (1) \qquad \frac{\partial B}{\partial t} = -\nabla \times E \quad (2) \qquad \nabla \cdot B = 0 \quad (3)$$

$$\frac{dV_i}{dt} = \frac{e_i}{m_i}(E(X_i) + V_i \times B(X_i)) \quad (4) \qquad \frac{\partial E}{\partial t} = \frac{1}{\varepsilon}(\nabla \times H - J) \quad (5) \qquad \nabla \cdot E = \frac{\rho}{\varepsilon} \quad (6)$$

Fig. 2. Particle-in-cell governing equations.

Computer Simulations. The grid in Fig. 1a is used to locate the particles and to project the effect of electromagnetic fields, charge density, and current density on the particles. Each block of the grid is known as a *cell*. That is where the name *particle-in-cell* comes from. Each cell has four vertex from which it is possible to perform several operations: interpolate the particle's position to project charge and current density into the grid, solve Maxwell's equations to obtain new values of electromagnetic fields, evaluate the changes on electromagnetic

fields and its influence onto the particles, push the particles to a new position with velocity changes, and affect the electromagnetic fields. This procedure is executed repeatedly and it is know as the *particle-in-cell loop*, represented in Fig. 1b. The PIC method is based on the aforementioned four operations known as [13]:

Scatter: the particle's position is calculated by using interpolation. The charge and current density is affected from each particle to each vertex. It has a different magnitude depending on the position of the particles; in other words, this operation calculates how the particles are affecting the grid.

Solve: once the grid is updated with the new values of charge and current density, the Maxwell's equations are calculated in every cell of the grid in order to obtain new values of the electric and magnetic fields forces.

Gather: the new values of the electric and magnetic fields are interpolated in the particle's location to calculate how are these affecting it. In an opposite way to Scatter, this phase reflects how the grid is influencing the particles.

Push: the changes on electromagnetic forces and its magnitude modify the speed of the particle pushing it to a new position. The Push phase is in charge of calculating the new speed and position of the particle.

The initialization phase of the Particle-in-cell algorithm consists in declaring a random position for all the particles with an initial speed of zero. Also, all the vertex of the grid should be initialized with a zero value for the electromagnetic fields. The finalization phase consists in reporting to the user the particles and grid information regarding positions, velocities, and values of electromagnetic fields. Using that information, scientists are able to obtain conclusions about the simulation done [13]. Based on the phases previously described, it is possible to create a pseudo code of the Particle-in-cell algorithm as presented by Algorithm 1.

Algorithm 1. Particle-in-cell High Level Algorithm

```
for each particle p ∈ N do                                  ▷ Initialization of particles
    X_p ← random
    V_p ← 0
for each vertex v ∈ Grid do                                 ▷ Initialization of fields
    E(v) ← 0
    B(v) ← 0
    J(v) ← 0
    ρ(v) ← 0
while t < t_max do
    ρ(v) ← ρ(v_{x,y}) + q_p · w(p, v_{x,y})                 ▷ Scatter: Particle interaction with the grid
    E(v_{x,y}) ← E'(v_{x,y})                                ▷ Solve: Calculate electromagnetic fields
    B(v_{x,y}) ← B'(v_{x,y})
    E(p) ← E(v_{x,y}) · w(p, v_{x,y})                       ▷ Gather: Reflect grid's influence to the particle
    X_p ← X'_p                                              ▷ Push: Change particle position
    V_p ← V'_p
Print X_p and V_p                                           ▷ Finalization of particle data
Print E, B, J, ρ                                            ▷ Finalization of grid data
```

PythonPIC. A Python implementation of the PIC method was made by Dominik Stańczak [16]. Called PythonPIC, the code models the interaction between a hydrogen plasma target and a laser impulse. The implementation of the Python code used Numba to improve performance, but the parallel programming functions of Numba were not used. In this paper, we use this code to extend it with parallel programming constructs and evaluate their impact on performance.

The code works in the following way. A configuration script describes the simulation that will be executed, along with all its parameters. This script uses a class named `Initial` from a configuration file, which interprets all the parameters and sets up the simulation. The `Initial` class inherits from `Simulation` in the `Classes` directory and uses functions described in the `Algorithms` and `Helpers` directories. The following is a detailed description of how the interaction of a laser with a hydrogen shield plasma is being implemented:

Configuration Scripts: The script `fulllaser.py` imports from `Configs` a file named `run_laser.py`. Several variables and the `initial` function are being imported. The configuration script uses this information to define the input parameters for `initial` function and execute.

Configs: The `run_laser.py` file imports file `BoundaryCondition.py` from Algorithms. From Classes imports `Simulation` and `Species` classes. From Helpers imports different functions and variables. This file describes the class `initial` which is inherited from `Simulation` and it is the one in charge of setting up everything to run the specific case of simulation that wants to be performed by using variables and functions from the files and classes previously mentioned.

Classes: This directory contains the three most important files of PythonPIC: `simulation.py`, `species.py`, and `grid.py`. These three files are used by all the Config files and are the ones in charge of handling and executing the simulation. The `Simulation` class takes all the information from the `Config` file and executes the desired simulation, creating the needed directories, initializing the Particle-in-cell grid, performing all the iterations, doing the post processing, and storing all the information. The `Species` class handles a set of particles and stores the information regarding position, velocity, and other variables. Finally, the `Grid` class handles the information regrading Particle-in-cell grid, like charges, currents, and fields for the particles in the simulation.

Algorithms: This directory contains files for the different algorithms used depending on the simulation case involved. File `BoundaryCondition.py` is being used to represent a boundary condition for the fields in the simulation. These files are in charge of mathematical calculations, and this could be a point to implement parallelization and even code optimization for specific simulation cases.

Helpers: This directory includes two files with different functions. File `helpers.py` has functions mainly for simulation progress configurations, while file `physics.py` includes common functions used in the simulation regarding mathematical calculations for the simulation.

2.2 Python Parallel Programming

Along with C and FORTRAN, Python is one of the most important programming languages in high performance computing. It comes at no surprise that the community has developed many Python tools for parallel programming. According to a recent study [11], there are more than 40 different Python parallel programming tools, each one with their particular combination of type of parallelism, execution mode and programming interface. Three of those tools that stand out for their simplicity and convenience at programming parallel code in Python are Multiprocessing [14], PyMP [10], and Torcpy [6].

Multiprocessing. It is a library that supports spawning processes with an API similar to any classic threading module, supporting local and remote concurrency. Originally, the Global Interpreter Lock (GIL) used by Python is in charge of scheduling the execution of threads, such that only one runs at a time. The Multiprocessing library avoids the limitations of GIL and uses sub-processes instead of threads. Therefore, it allows the use of multiple processors [14,18]. There are two basic ways to exploit parallelism using the multiprocessing library: *Pools* and the *Process class*. The usage of Pool is intended for the execution of one function for multiple input values, distributing the input data across different processes. On the other hand, by using the Process Class, the processes are spawned by creating an object and then calling a start and join methods. These two methods, specially the Process class, are the base to start making parallel code using the multiprocessing library. Below, you will find a code sample for a parallel sum of two arrays.

```
1  from multiprocessing import Pool, Array
2  def sum(i):
3      c[i] = a[i] + b[i]
4  if __name__ == '__main__':
5      global a, b, c
6      N = 5
7      a = [1,2,3,4,5]
8      b = [2,4,6,8,10]
9      c = Array('f', range(N))
10     with Pool(4) as p:
11         p.map(sum, range(N))
12     print(c)
```

PyMP. Build on top of Multiprocessing library, PyMP is a Python library that offers parallel programming functionalities in the style of OpenMP. It takes the small code changes and high efficiency of directive-based programming and combines it with Python usage easiness [10]. Since pragmas are not present in Python, PyMP leverages some language constructs to provide parallel programming features. The `with` statement provides parallel contexts for several threads. The `range` instruction divides loop iterations among active threads. Other configuration options (number of threads, loop scheduling policies, thread-specific identifiers, variable scope) are passed as parameters to functions. Only a portion

of the OpenMP standard can be mapped to PyMP language constructs. However, the available functionalities are powerful to represent a modest range of parallel algorithms. Below is the PyMP version of the parallel sum of two arrays.

```
import pymp
if __name__ == '__main__':
    N = 5
    a = [1,2,3,4,5]
    b = [2,4,6,8,10]
    c = pymp.shared.array(N, dtype='float64')
    with pymp.Parallel(4) as p:
        for i in p.range(N):
            c[i] = a[i] + b[i]
    print(c)
```

Torcpy. It is an open source library supported by IBM that provides a parallel computing framework with a unified approach for expressing and executing task and data parallelism on both shared and distributed memory architectures [6]. Although it uses MPI internally in a transparent way to the user, Torcpy also allows the use of explicit MPI code at the application level. It provides support for parallel nested loops, map functions, and task stealing at several levels of parallelism. The `submit` and `wait` functions provide the necessary task parallelism operations, while `map` function implements data parallelism. Below is the Torcpy version of the parallel sum of two arrays.

```
import torcpy as torc
def sum(i, a, b):
    return a + b
def main():
    N = 5
    a = [1,2,3,4,5]
    b = [2,4,6,8,10]
    iterations = range(N)
    c = torc.map(sum, iterations, a, b)
    print(c)
if __name__ == '__main__':
    torc.start(main)
```

2.3 Related Work

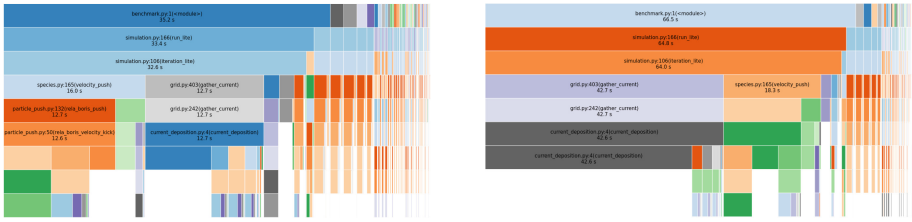
Python implementations of the Particle-in-cell method are easy to find in the available literature. Blandón et al [2] presents a one-dimensional PIC implementation using Anaconda packages. They use their sequential code to study plasma phenomena, such as oscillations, waves, instabilities and damping. Fink et al [3] used a PIC code to compare two parallel programming tools in Python (Charm4Py and mpi4Py). They started with an already parallel MPI version of the code and ported it to parallel objects [4]. Their results highlight the scalability of both approaches on distributed-memory systems, with parallel objects providing an advantage in load imbalanced scenarios. Kadochnikov [7] accelerated a PIC implementation in Python on GPUs, using CUDA through CuPy library. The code in that paper used algebraic multigrid solvers in Python to create a code able to understand some instabilities in electron beam ion sources.

There are previous works comparing tools for parallel programming in Python. Adekanmbi et al [1] implemented a solution to the N-body problem using three different HPC Python tools: Taichi, Numba, and NumPy. The former two provide the shortest execution time, since both are based on a just-in-time compiler. Kim et al. [8] surveyed parallel processing tools in Python and provided experimental results showing the advantages of a couple of tools (Pandaral-lel and Ipyparallel). Using those tools on a multi-core chip, they obtained 5.2x and 2.6x speedups, respectively. Miranda and Stephany [12] used a five-point stencil program to compare HPC Python tools (Cython and Numba) against a reference implementation in FORTRAN. Experimental results show the FORTRAN and F2Py versions are marginally faster than their Python counterparts. Therefore, Python provides a competitive alternative to traditional programming languages for HPC.

3 Implementation

3.1 Profiling

Prior to start any code modification, it is necessary to understand how the code is behaving from the time consumption perspective. The code profiling indicates which are the most time consuming functions in the execution. By understanding these functions, it is possible to prioritize them for parallelism purposes, a reduction of the execution time of these functions is more significant for the global execution time.



(a) Original Code Profiling. (b) For Loops Profiling.

Fig. 3. Profiling of PythonPIC code.

Two profiles were created as presented in Fig. 3. In Fig. 3a the original code was profiled. It was possible to notice two main functions consuming the majority of the execution time `rela_boris_velocity_kick` and `current_deposition`, these are part of the Particle-in-cell method. The first function was subdivided into several functions, while `current_deposition` was not subdivided, meaning this is the most time consuming function. There, multiple vectorized operations were observed. These were converted to for loops and a second profile was done presented in Fig. 3b. The `current_deposition` function now takes longer to execute, but the conversion from vectorized functions to for loops is necessary to implement code parallelism. The converted function is a multiplication of arrays, each array has the size of the amount of particles in the simulation.

3.2 Code Transformation

The code is freely available through the following Git repository:
https://github.com/oblandino/PythonPIC/

Multiprocessing. The implementation's structure was developed as presented in the Background section, for a multiplication of arrays. Multiprocessing is part of Python standard libraries, no extra packages were required to be installed in order to use it. The map function only admits two arguments, the function and the iterable, so the arrays were required to be declared as global variables in order to be used by the external function performing the array multiplication. The array that stores the information had to be declared as a multiprocessing `Array`, in order to be shared by the workers and store information in parallel. The `map` function allows a third argument, the chunksize, its function is to split the iterable into chunks specified by the number of chunksize, the default value is 1. This is used to improve efficiency, as well as the `imap` function, which was used in the code due to the large number of iterations.

PyMP. Based on the for loops code, PyMP was very easy to implement because the required changes are minimal, as presented in the background. In PyMP, it was also required to declare the array as a PyMP shared array in order to be shared by the workers. PyMP does require external package installation, but overall this was the easiest implementation.

Torcpy. This implementation was done using a `map` function similar to Multiprocessing. The main difference is that Torcpy does not handle global variables, instead it allows all the required arguments in the `map` function. For this reason, the arrays were not declared as global, instead these were given as arguments to the `map` function, and the external function performing the array multiplication had them as inputs. Torcpy requires a one time initialization by using a `start(f)` function were `f` is the function that includes the Torcpy instructions. For PythonPIC, due to the code implementation and amount of iterations, the `start()` function was required to be integrated in the `simulation.py` file, initializing the parallel environment sooner than Multiprocessing and PyMP, potentially causing overhead. Torcpy allows the chunksize argument as presented in Multiprocessing.

4 Experimental Results

4.1 Setup

All experiments in this paper were run on *Kabré* supercomputer at the National High Technology Center (CeNAT) of Costa Rica. Kabré is a hybrid compute cluster comprising a total of 52 computing nodes of 4 different architectures. Two of those architectures are relevant for the experiments below. First, the *manycore*

Table 1. Software configuration.

Program	Version
Operating system	CentOS
OS distribution	7.9.2009
OS kernel	3.10.0-1160.62.1.el7.x86_64
Python	3.9.7
Multiprocesing library	Python 3.9.7
PyMP library	0.5.0
Torcpy library	0.1.1
Numba library	0.55.1
cProfile	1.0.7
SnakeViz	2.1.1

Table 2. PythonPIC parameters.

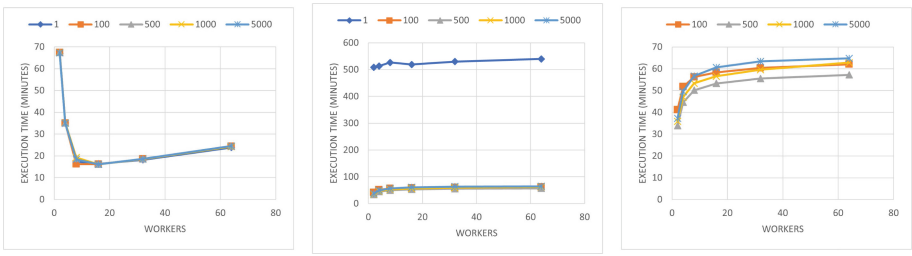
Item	Value
Number of particles	100000, 200000, 400000
Number of iterations	584
Number of trials	10

nodes have each an Intel Xeon Phi KNL 7230 processor, running at 1.30GHz. Each node has 96 GB of main memory. Second, the *multicore* nodes contain an Intel Xeon Gold 6354 processor, running at 3.00GHz. Each node has 512 GB of main memory. Kabré is interconnected with an Ethernet 10Gb network and runs Linux CentOS operating system. Table 1 summarizes the configuration of the software stack used for the experiments. The execution time parameters for PythonPIC are presented in Table 2. Only average results with a coefficient of variation lower than 3% are plotted in the experiments below.

4.2 Experiments

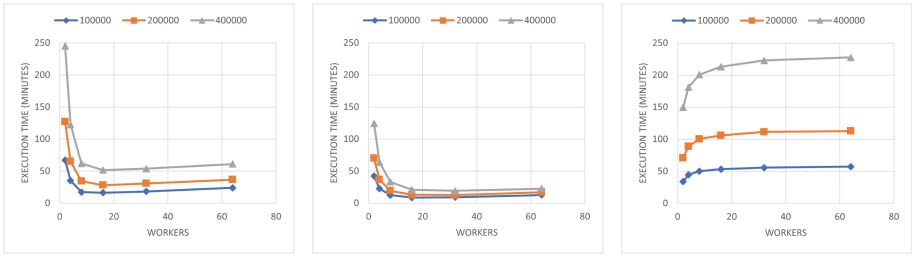
Manycore Processor. After the code was transformed, there was still a missing piece for Multiprocessing and Torcpy, the chunksize. In the documentation of both tools, there is no specification on how to define this parameter. The developers suggest to use a *large* value, but also mention that a *very large* value can actually cause overhead and memory inefficiencies. Figure 4 shows the results obtained in a 100,000 particle simulation. Figure 4a shows that the changes in chunksize did not affect the overall behavior of Multiprocessing. For Torcpy, Fig. 4b shows a difference of around 20x between using the default value against other selected chunksize values. Figure 4c is named Torcpy Reduced, because the default value was removed to provide a better scale, the best execution time was obtained with a chunksize value of 500. In any of the cases a time reduction was observed, meaning that Torcpy does not scale properly in the manycore processor.

The default chunksize value was used for Multiprocessing, and a chunksize value of 500 for Torcpy. Figure 5 presents the results obtained for a strong-scale experiment. The best results were obtained with PyMP, then Multiprocessing, and lastly Torcpy. In Figs. 5a and 5b, for Multiprocessing and PyMP respectively, the best results were obtained by using 16 workers, a greater value introduced overhead and the results started to slowly increase. The best execution times were presented by PyMP. On the other hand, as it was expected for Torcpy, a time reduction was not observed in Fig. 5c. On the contrary, execution times increased as the number of workers increased.



(a) Multiprocessing (b) Torcpy (c) Torcpy Reduced

Fig. 4. Manycore processor chunksize comparison.

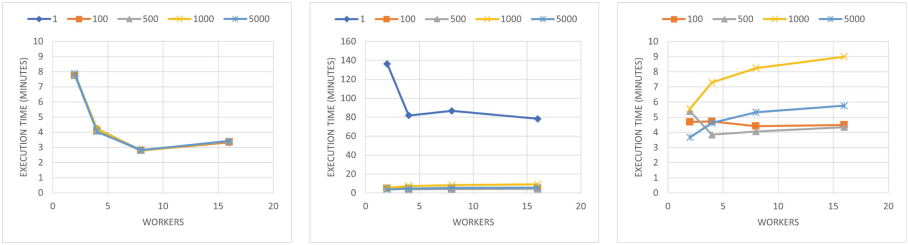


(a) Multiprocessing (b) PyMP (c) Torcpy

Fig. 5. Manycore processor strong-scaling results.

Multicore Processor. A similar approach was performed for the multicore processor, the chunksize selection was the first step to follow using the same variables and values presented before. The results presented in Fig. 6 were obtained in a 100000 particles simulation. Figure 6a shows that changes in chunksize value did not affect the overall behavior of Multiprocessing implementation. Figure 6b presents differences around 20x between using the default value of chunksize, against other selected values for Torcpy. To provide a better scale, the default chunksize value was removed as presented in Fig. 6c also named Torcpy Reduced, the best reduction of execution time was obtained with a chunksize value of 500, greater values were causing overhead.

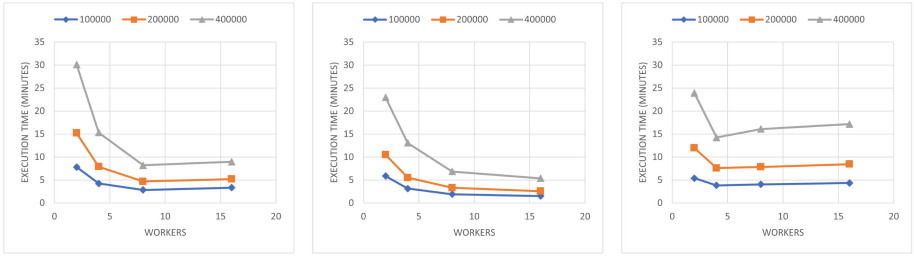
Using the default chunksize value for Multiprocessing, and a chunksize value of 500 for Torcpy, the results presented in Fig. 7 were obtained. Similar execution times were observed, but the best results were presented by PyMP, then Multiprocessing, and lastly Torcpy. Not only the execution times were lower using PyMP, but also the scalability of workers was better. The behavior for Multiprocessing presented in Fig. 7a was similar to the one presented by PyMP in Fig. 7b. The main difference, besides execution time, was that by using 16 workers Multiprocessing showed an increase of execution time, while PyMP presented the expected reduction. Figure 7c shows that the reduction of execution time for



(a) Multiprocessing (b) Torcpy (c) Torcpy Reduced

Fig. 6. Multicore processor chunksize comparison.

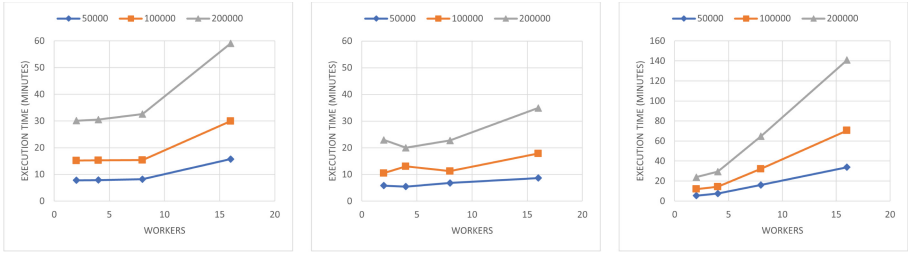
Torcpy was only by using 4 workers, a greater value presented a slow increasing behavior which was more notorious in the 400,000 particles simulation.



(a) Multiprocessing (b) PyMP (c) Torcpy

Fig. 7. Multicore processor strong-scaling results.

Weak-Scaling results are presented in Fig. 8 for the multicore processor, these present the number of particles per worker for the simulation. The expected case is to obtain a line with slope equals to zero, this would mean that as the particles and number of workers increase, the execution time remains the same, indicating perfect scalability. The best results were presented by PyMP in Fig. 8b, the slope does not increase as fast as in Multiprocessing in Fig. 8a or Torcpy in Fig. 8c. The execution times were also shorter. It is important to highlight that the slope was almost zero for Multiprocessing when using 8 workers or less, in contrast to PyMP which had more variation, even when it got better overall results.



(a) Multiprocessing (b) PyMP (c) Torcpy

Fig. 8. Multicore processor weak-scaling results.

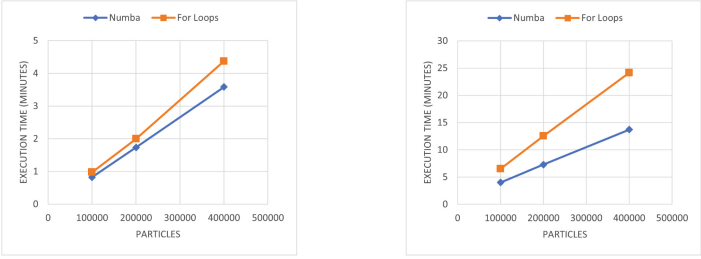
5 Discussion

Several parallel programming tools for Python have appeared in the last decade. Presumably, that is a consequence of the original language specification of Python not including native constructs for parallelism. A survey about Python tools for HPC found more than 40 libraries [11]. Those libraries come in all flavors, some mirroring parallel computing paradigms in other languages, some offering supposedly Pythonic alternatives. Also, the set of libraries contain efforts already deprecated, while others are still active. This plethora of options offer an interesting environment for exploring the advantages and downsides of each alternative.

This paper compared three libraries for parallel programming in Python and targeting symmetric multiprocessors. The first option, Multiprocessing, offers generality as it provides features of both task and data parallelism. Performance is competitive and maintenance of the library is active. The second option, PyMP, gives good performance and ease of programming. It truly resembles the basic features of the OpenMP standard in other programming languages. That characteristic facilitates the adoption of this library in a community exposed to traditional shared-memory programming paradigms. The third option, Torcpy, provides a very neat interface for doing both data and task parallelism. Its performance is still lacking, but it has the backup of a legendary company in the world of HPC.

A natural question after examining a group of parallel programming libraries in Python relates to their relative performance compared to a just-in-time (JIT) compiled alternative. To complement the results of this paper, we added an experiment with a PythonPIC implementation that includes Numba instructions. Numba is an open-source JIT compiler that uses LLVM to translate a subset of Python into machine code. Figure 9 shows the result of running and reporting average execution time of 10 repetitions. The plots in 9a and 9b offer the performance results in the multicore and manycore processor, respectively. In both nodes, the execution time is reduced when using Numba, the delta increases as the number of particles also increase. The best performance is observed in the

manycore processor showing a reduction of 43% in the execution time of the 400,000 particle simulation.



(a) Multicore Processor (b) Manycore Processor

Fig. 9. Numba experimental results.

6 Final Remarks

The particle-in-cell method has established itself as one of the cornerstones for understanding physical phenomena in a variety of domains, particularly in plasma physics. As scientists extend this method and increase the granularity of the simulation, the computational requirements dramatically grow. Inevitably, high performance computing resources are needed to execute the simulations and push the envelope in scientific research.

Along with FORTRAN and C/C++, Python is a popular programming language for scientific computing and HPC. The broad adoption of Python in scientific communities, makes it an appropriate alternative for scaling applications. There are several Python tools for parallel programming, with some of them focused on symmetric multiprocessor architectures. Hence, it is important to compare those tools according to the programming features they provide and the performance they show.

PyMP obtained better performance results compared to Multiprocessing and Torcpy, its execution time was shorter, its scalability to increasing workers was better, and its implementation was easier. Multiprocessing presented a similar behavior than PyMP, but the execution times were longer. In both architectures analyzed the best results were obtained by using 16 workers. In the multicore processor, Torcpy presented better execution times than Multiprocessing when using 2 and 4 workers, a greater value did not scale properly. In the manycore processor, Torcpy never presented a time reduction. Multiprocessing and Torcpy require a characterization of the chunksize value depending on the problem involved when the map function is used. The behavior of the multicore processor changed depending on the value used. Numba is an excellent option to use depending on how the code is implemented.

Acknowledgments. This research was partially supported by a machine allocation on Kabré supercomputer at the Costa Rica National High Technology Center.

References

1. Adekanmbi, O.G.: Performance comparisons for Python libraries in parallel computing and physical simulation. In: 2022 ASEE Gulf Southwest Annual Conference. ASEE Conferences, Prairie View, Texas, March 2022. https://peer.asee.org/39194
2. Blandón, J.S., Grisales, J.P., Riascos, H.: Electrostatic plasma simulation by particle-in-cell method using ANACONDA package. J. Phys. Conf. Ser. **850**, 012007 (2017)
3. Fink, Z., Liu, S., Choi, J., Diener, M., Kale, L.V.: Performance evaluation of Python parallel programming models: charm4Py and mpi4py (2021). https://doi.org/10.48550/ARXIV.2111.04872, https://arxiv.org/abs/2111.04872
4. Galvez, J.J., Senthil, K., Kale, L.: CharmPy: a Python parallel programming model. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 423–433 (2018). https://doi.org/10.1109/CLUSTER.2018.00059
5. Fehske, H., Schneider, R., Weiße, A.: Computational Many-Particle Physics. Springer, Cham (2008)
6. (IBM) Integrated Baseboard Management Controller (iBMC) : torcpy: supporting task-based parallelism in Python (2019). https://github.com/IBM/torcpy
7. Kadochnikov, I.: Accelerating the particle-in-cell method of plasma and particle beam simulation using CUDA tools. In: 27th International Symposium on Nuclear Electronics and Computing (NEC 2019) (2019)
8. Kim, T., Cha, Y., Shin, B., Cha, B.: Survey and performance test of python-based libraries for parallel processing. In: The 9th International Conference on Smart Media and Applications, SMA 2020, pp. 154–157. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3426020.3426057
9. Lapenta, G.: Kinetic plasma simulation: particle in cell method
10. Lassner, C.: PyMP (2016). https://github.com/classner/pymp
11. Meneses, G.C.S.E.: Parallel programming tools in Python
12. Miranda, E., Stephany, S.: Comparison of high-performance computing approaches in the python environment for a five-point stencil test problem. In: Anais do XV Brazilian e-Science Workshop, pp. 33–40. SBC, Porto Alegre, RS, Brasil (2021). https://doi.org/10.5753/bresci.2021.15786, https://sol.sbc.org.br/index.php/bresci/article/view/15786
13. Pous, X.S.: Particle-in-cell algorithms for plasma simulations on heterogeneous architectures
14. Python: multiprocessing - process-based parallelism. https://docs.python.org/3/library/multiprocessing.html
15. Rao, V.R.: Here's why you should use Python for scientific research (2018). https://developer.ibm.com/blogs/use-python-for-scientific-research/
16. Stańczak, D.: Implementation and performance analysis of particle-in-cell simulation software in Python (2017)
17. Top500: Top500 list. Top500 (2022). https://www.top500.org/
18. Zetcode: Zetcode. https://zetcode.com/python/multiprocessing/