10.18605/2175-7275/cereus.v13n2p84-98

ARTIGO ORIGINAL

<< Recebido em: 23/03/2021 Aceito em: 16/06/2021. >>

REVISTA CEREUS

ISSN: 2175-7275

Common HPC Approaches in Python Evaluated for a Scientific Computing Test Case

Abordagens comuns de HPC em Python avaliadas para um caso de teste de computação científica

Eduardo F. Miranda[1], Stephan Stephany[2]

[1] Graduate Program in Applied Computing (CAP) / National Institute for Space Research (INPE). São José dos Campos, SP - Brazil.

E-mail: eduardo.miranda@inpe.br

[2] Coordination of Applied Research and Technological Development (COPDT) / National Institute for Space Research (INPE). São José dos Campos, SP - Brazil.

E-mail:

stephan.stephany@inpe.br

ABSTRACT

A number of the most common high-performance computing approaches available in the Python programming environment of the LNCC Santos Dumont supercomputer, are compared using a specific test case. Python includes specific libraries, development tools, implementations, documentation and optimizing/parallelizing resources. It provides a straightforward way to program in a high level of abstraction, but parallelization resources to exploit multiple cores, processors or accelerators like GPUs are diverse and may be not easily selectable by the programmer. This work makes a comparison of common approaches in Python to boost computing performance. The test case is a well-known 2D heat transmission problem modeled by the Poisson partial-differential equation, which is solved by a finite difference method that requires the calculation of a 5-point stencil over the domain grid. Their serial and parallel implementations in Fortran 90 were taken as references in order to compare their performance to some serial and parallel Python implementations of the same algorithm. Besides performance results, a discussion about the trade-off between easiness of programming versus processing performance is included. This work is a primer for the use of HPC resources in Python.

Keywords: High performance computing, Python language, Finite-difference method, Scientific computing, Performance comparison

1. INTRODUCTION

Python is a modern and user-friendly language, featuring an easy syntax, good readability, easy interfacing with external applications, fast implementation using scripting, access to a wide community of developers, and with a huge collection of libraries, scientific or not (LUNACEK; BRADEN; HAUSER, 2013; VIRTANEN et al., 2020). Furthermore, Python supports High Performance Computing (HPC) by means of embedded or external libraries (SEHRISH et al., 2017). A powerful programming environment is provided by combining Python with an interactive shell like IPython, allowing fast prototyping. According to the IEEE Spectrum programming language ranking (IEEE SPECTRUM, 2020), Python is the most popular.

The use of Python is also widespread for scientific applications, for instance, at INPE (the Brazilian National Institute for Space Research), where its digital library lists 88 references for this language, including diverse applications such as the optimization of a mathematical model for estimating the amount of incident Solar radiation in the Earth surface and the use of a neural network for the classification of supernovae. Python is available in many compiler packages like Intel and in most of the supercomputer programming environments. Application programs implemented in languages like Fortran 90 or C, which typically demand massive parallel processing, can be encapsulated in the Python environment by means of wrappers, in a modular way. Such flexibility makes easy to perform simulations, data analysis and visualization (BEAZLEY; LOMDAHL, 1997), mainly for large scale scientific applications. Thus, Python provides an interactive, user-friendly programming environment that is convenient to trial-and-error, greedy, or other exploration schemes, common in scientific computing (HINSEN, 1997).

This work aims at the exploration of the most common parallelization approaches that are available in the Python ecosystem, which includes libraries, frameworks, tools, etc. It intends to described and discuss some High-Performance Computing (HPC) approaches that are available in Python programming like parallel processing, or general-purpose graphics processing unit (GPGPU or simply GPU) processing. The performance of these Python HPC approaches is then compared to the correspondent serial and parallel F90 implementations for a specific test case, a finite-difference method to solve the partial differential equations resulting from the Poisson equations for a 2D heat transfer problem. There is a trade-off between languages like F90 or C and the Python environment concerning the easiness of programming and the processing performance. Such languages are harder to implement an application than Python, but are straightforward to optimize/parallelize and provide better performance. However, there are nowadays many libraries and frameworks that provide HPC resources for Python, making difficult to analyze such trade-off.

In short, this work compares common HPC approaches in Python, by means of a given test case. It is also intended to provide a guidance for Python programmers looking for HPC resources. It compares serial and parallel performance of different implementations of a 5-point stencil test case related to a 2D heat transfer problem. These implementations were coded in Fortran 90 (henceforth referred to as F90) and in the most common HPC Python choices that were selected based on their processing performance. Except for a one GPU implementation, parallelization is achieved by use of the message passing communication library MPI (Message Passing Interface) (GROPP et al., 1996). Some general considerations about this work follows in order to explain its scope in the vast Python environment:

1. Python environment is very diverse and Python code can be linked to a multitude of APIs/libraries for HPC, and thus programs can be written in different ways;
2. Python implementations of this work include HPC solutions for standard Python, Cython, Numba, Numba-GPU and F2Py, but there are many others;
3. Python multiprocessing environment allows any parallel execution, from MPI processes to Open MP threads, using a personal laptop/PC or supercomputer, but in this work, the different HPC implementations were based on MPI for Python, except for Numba-GPU; executions were performed in one or more nodes of a supercomputer;
4. Performance results are particular of the selected test case and its problem size, and thus it can be expected that different applications and problem sizes may lead to a different analysis of the processing performance, but may help the Python programmer to choose a more convenient Python HPC implementation.

This work is a short primer for the use of HPC resources in the Python programming environment, using the LNCC Santos Dumont supercomputer [1], henceforth referred as SDumont.

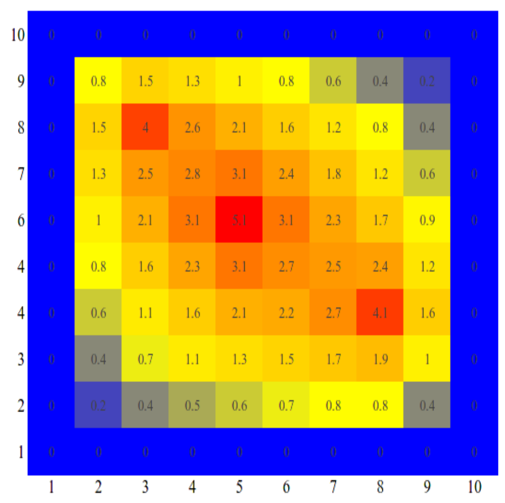
[1] https://sdumont.lncc.br/machine.php?pg=machine#

2. THE FIVE-POINT STENCIL TEST CASE

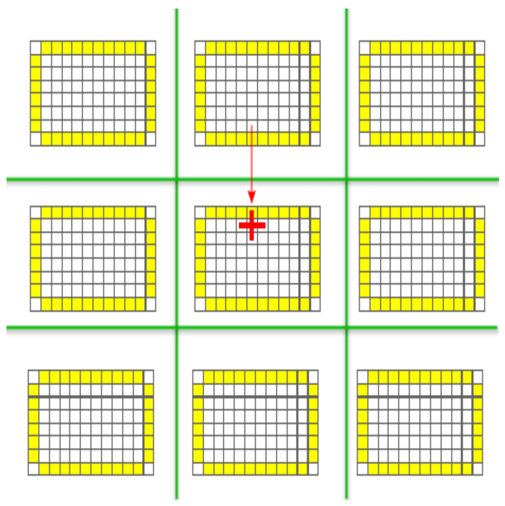
Processing performances of the Python implementations were evaluated, taking as reference the serial and parallel F90 ones. The adopted test case is a well-known heat transfer problem over a finite surface (Figure 1), modeled by the Poisson partial-differential equation. It models the normalized temperature distribution over the surface along a number of iterations that compose the simulation. As commonly employed for numerical solutions, this equation is discretized in a finite grid and solved by a finite difference method. The specific algorithm requires the calculation of a 5-point stencil over the 2D domain grid (CHEN et al., 2002) in order to update the temperatures at every time step. A uniform temperature field with zero value is assumed over the surface, and typically, adiabatic or Dirichlet boundary conditions are assumed, being the latter assumed for this problem. Three constant-rate heat sources were placed in localized grid points, each one inputs a unit heat quantity every time step. The heat transfer simulation is modeled by a finite number of time steps, being all grid points updated at every time step. The temperature distribution will be determined by the heat sources and the Dirichlet boundary conditions, which implies in zero temperature at the border grid points. The 5-point stencil consists in updating a grid point by averaging the temperatures of itself with the temperatures of its four neighbors, left-right and up-down in the grid. The temperature field U is defined over a discrete grid (x,y) with spatial resolutions $\Delta x = \Delta y = h$. Thus, the discretization maps real Cartesian coordinates (x,y) to a discrete grid (i,j) , with $U_{x,y} = U_{i,j}$, $U_{x+h,y} = U_{i+1,j}$ for the x dimension, and analogously for the y dimension. Therefore, the discretized 2D Poisson equation with a 5-point stencil is expressed by Equation 1.

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} \approx \frac{U_{i+1,j} + U_{i,j+1} - 4U_{i,j} + U_{i-1,j} + U_{i,j-1}}{h^2} \qquad (1)$$

Considering the chosen test problem, an initial implementation of the algorithm, proposed by Balaji (BALAJI et al., 2017) in the C language, was easily ported to F90. The discretized 2D domain of the heat transfer problem chosen as the test problem showing 9 sub-domains divided by green lines, with their grid points, and their phantom zones in yellow, is shown to the right of Figure 1, where the red cross denotes the 5-point stencil. The left side of Figure 1 shows the final temperature distribution over a finite surface after 500 iterations, exemplified by the grid 10 x 10 and three arbitrarily chosen heat sources, shown as red cells. The simulation covers only the internal grid 8 x 8, and the initial zero temperature distribution is indicated in blue.



(a) Final temperature distribution.



(b) Discretized 2D domain. *Note*. Adapted from (BALAJI *et al.*, 2017).

Figure 1: Initial and final temperature distribution over a finite surface.

3. F90 AND PYTHON IMPLEMENTATIONS OF THE TEST CASE

Several serial and parallel implementations of the test case described in the previous section, have been written and executed. Their description and processing performance results are shown ahead. Implementations include F90 (SOUZA; PANETTA; STEPHANY, 2018), taken as reference, standard Python, Fortran to Python (F2Py), Cython, and Numba (including Numba-GPU). In this work, all parallel versions of these implementations are based on the Message Passing Interface (MPI) communication library, which is wrapped by the MPI for Python (mpi4py) (DALCÍN et al., 2008) and allow the execution of MPI processes from the Python environment. Parallelization using threads was restricted to the execution of a Numba-compiled Python function in a GPU, which is also discussed ahead. The mpi4py parallelization allows to employ a single or many computer nodes.

3.1. F90 SERIAL AND PARALLEL

F90 is described because it was taken as a reference for Python implementations, and was compiled with GNU gfortran. The serial version is the implementation of the algorithm described in the previous section, and the parallel version employs the standard MPI asynchronous non-blocking communication functions MPI_Isend( ) and MPI_Irecv( ). At the end of each time step of the algorithm, synchronization is necessary for each sub-domain to update the phantom zones of all neighboring sub-domains. For a square grid with N x N points and p MPI processes, each process is assigned a sub-domain with a total of [(N/p)+2]x[(N/p)+2] points. The part of the code that requires performance, updates the domain grid using the 5-point stencil, as shown in Listing 1.

Listing 1: Time consuming part of the test case F90 code.

```
do j = 2,by+1
  do i = 2,bx+1
    anew(i,j)= 1/2*(aold(i,j) + 1/4*(aold(i-1,j) + aold(i+1,j) + aold(i,j-1) + aold(i,j+1)))
  enddo
enddo
```

3.2. STANDARD PYTHON SERIAL AND PARALLEL

The portability of the F90 code to Python is straightforward, without any external library except the NumPy library of numerical tools. Part of the Python loops can be executed transparently by resources from the NumPy library. The structure and sequence of the original code are preserved and executed interactively by the interpreter Python in the Jupyter Notebook environment. In this way, it is easy to modify the code or its parameters, show and record the results, perform the prototyping to check the accuracy of the algorithm and document its code and description. In this step, the user can take advantage of the modular nature of Python to selectively optimize the code, for example, by porting a specific module to F90 or replacing it with a library function for performance. Parallelization is possible using the Python multiprocessing environment which provides many different ways of providing parallel execution, according to the chosen library. However, in this work, the library mpi4py (MPI for Python) was used.

3.3. F2PY SERIAL AND PARALLEL

F2Py is a wrapper that creates a Python module by compiling F90 source code, and requires the definition of a function with a list of arguments to be passed for the execution of the module Python, such as number of grid points, location and heat rate of sources, number of iterations, etc. In the case of the F90 code already paralleled with MPI, the module can be executed in parallel by the F90 code itself. However, if the F90 code is not yet parallelized, the typical alternative is to use the library mpi4py (MPI for Python), but eventually it will be necessary to break the original code into parts and choose computationally intensive ones to parallelize with MPI. Therefore, F2Py seems more convenient when different modules Python need to be generated and called interactively.

The mpi4py library resembles the F90 MPI and allows Python to perform a job using one or more shared memory computer nodes, each with multicore processors.

3.4. CYTHON SERIAL AND PARALLEL

Cython is a static optimization compiler for the programming languages Python and Cython. It is usually used to create standard Python modules, and for wrapping code C/C++. The source code in Cython is compiled first to the language C/C++, which is then compiled transparently by the operating system's standard C/C++ compiler, generating executable machine code. Each module generated by Cython requires the definition of a corresponding standard Python function, and the performance depends on the syntactic and semantic extensions used, resources used of the Python interpreter, and choice of libraries. The result of the compilation is a Python library that includes an API for the generated code. In the case of the code for the serial test problem, the complete code was compiled by Cython, while in the case of the parallel code, a separate module was created compiling the double loop that updates the 2D domain, and then the mpi4py was used. The time-consuming function that updates the domain grid using the 5-point stencil is shown in Listing 2.

Listing 2: Time consuming part of the test case Cython code.

```
cpdef stp(double[:,::1] anew, double[:,::1] aold,Py_ssize_t by,Py_ssize_t bx) :
  for i in range(1,bx+1) :
    for j in range(1,by+1) :
      anew[i,j] = 1/2*(aold[i,j] + 1/4*( aold[i-1,j] + aold[i+1,j] + aold[i,j-1] + aold[i,j+1]))
```

3.5. NUMBA SERIAL AND PARALLEL

Numba is a JIT (just-in-time) compiler, which compiles the code at run time. Numba uses part of the Python language and part of the NumPy module. It uses resources from the LLVM project which is a collection of modular and reusable compilers and toolchain technologies developed by the University of Illinois in Urbana-Campaign since 2000, that allows to generate machine code optimized for CPU or GPU. To compile the code for the test case, an approach similar to Cython was adopted, by creating a Python function that embeds the part of the code, and is compiled by Numba. The rest of the Python code is interpreted by standard Python, as it runs fast. In the case of parallel code to be executed in multiple cores, the mpi4py library is used, and execution using GPU is also possible, since Numba supports part of the Nvidia CUDA API, requiring the definition of the kernel that will run on the GPU. Processing performance for the Numba-GPU version is shown below in Section 4.7. The time-consuming function is shown in Listing 3.

Listing 3: Time consuming part of the test case Numba code.

```
@jit(nopython=True)
def kernel(anew,aold):
  anew[1:-1,1:-1]=1/2*(aold[1:-1,1:-1]+1/4*(aold[2:,1:-1]+aold[:-2,1:-1]+aold[1:-1,2:]+aold[1:-1,:-2]))
```

4. ANALYSIS OF THE PARALLEL PERFORMANCE FOR THE TEST CASE

Standard HPC performance metrics are used here like the speedup S_p, given by the ratio between the serial execution time t_s and the parallel execution time t_p, using p processes or threads ($S_p=t_s/t_p$), being the ideal speedup, called linear speedup, equal to p.

In this work, for the considered code, the speedup is calculated using the serial implementation and the corresponding parallel implementation, i.e. same compiler and language. However, an overall ranking of speedup and efficiency is also shown, in order to compare the different implementations. Another standard metric is the parallel efficiency E_p, given by the ratio between the speedup and the corresponding number p of processes or threads ($E_p=S_p/p$). Thus, the linear speedup yields the unitary efficiency.

4.1. COMMENTS ON THE PERFORMANCE OF THE CPU IMPLEMENTATIONS

This section shows the serial and parallel processing performance of the implementations executed only in CPUs, i.e. in one or in multiple processor cores of one or more computer nodes, without any GPU. **Erro! Fonte de referência não encontrada.** shows processing times of the test case for the different implementations in one or more SDumont Bull B710 computer nodes. All processing times shown here are the average of 3 executions of each implementation for the different number of processes. The same table also shows processing times for the serial and for the MPI version with 1, 4, 9, 16, 36, 49, 64, and 81 processes in the Bull B710 nodes. Figure 2 shows speedups and efficiencies calculated taking the F90 serial time as reference.

In general, according to **Erro! Fonte de referência não encontrada.**, the F90 and the F2Py achieved the best performance, with the latter yielding the lowest processing time of 1.01 s with 81 MPI processes. They are followed by the Cython and Numba implementations, with standard Python well behind.

F2Py requires little changes to the F90 original code, while Cython and Numba, little changes to the Python code. However, Numba performance was comparable to the others only from 4 up to 36 processes. The very poor performance of the standard Python serial and parallel versions shows the convenience of using implementations like F2Py, Cython or even Numba. Once the reference time for speedup calculation was the same F90 serial time for all implementations, the rankings for speedups and parallel efficiencies are similar to the ranking for processing times.

As can be seen in Table 1 and Figure 2, parallel scalability is poor, since the algorithm of the test case updates all grid points at every time step thus requiring the exchange of border temperatures between neighboring sub domains in order to update the corresponding ghost zones. This update demands an amount of communication between processes that drops the parallel efficiency below 40% for 9 MPI processes or more, in the case of most implementations shown here. It can be seen that for up to 36 MPI processes, executed in two Bull B710 nodes, all implementations performed similarly, except for standard Python. In the case of 81 MPI processes, which require 4 computer nodes, the performance of all implementations suffered a significant drop in comparison with 64 MPI processes. The exception was F2Py that obtained the lowest time with 81 processes.

4.2. COMMENTS ON OBTAINED F90 SERIAL AND PARALLEL PERFORMANCE

The parallelization was performed by dividing the domain of the test case into square sub-domains of up to 9 x 9 that is performed by 81 processes. The F90 implementation, sequential, was used as a reference for evaluating the other sequential implementations, for execution in a core, without parallelization. The implementation does not depend on Python and uses a Fortran compiler. It uses as execution parameters, the number of points of the grid, the energy to be inserted, and the number of iterations. The results are evaluated in the Jupyter Notebook interactive environment, using the magic command writefile to write the source code without changes, and the magic command bash is used to access the shell and on the command compile and create the executable file that will be sent to an execution node through the queue manager and scheduler submission file. Scheduler commands such as sbatch and squeue, are executed in the Jupyter Notebook environment, and the output files resulting from the execution are also read and evaluated on the Notebook. In this way the process is documented and is reproducible using the notebook itself. F90 obtained the best sequential execution time among the implementations, however in the parallel execution the results varied, and for 16, 49, and 81 processes, F90 did not obtain the best time. As a suggestion for a future study, it would be interesting to investigate why F90 did not get the best time under these conditions.

4.3. COMMENTS ON OBTAINED PYTHON SERIAL AND PARALLEL PERFORMANCE

The sequential implementation in Python is a direct translation of the original code, without any external library other than NumPy. In general, it preserves the same structure and execution sequence as the original code, and computationally intense repetition loops are performed by NumPy. As the execution is interpreted, the execution time is high. This type of implementation serves, for example, as a proof of concept developed in rapid prototyping, and runs practically unchanged on different platforms, relying only on a standard Python interpreter. Since Python is an easy-to-read language, especially when used in interactive computing environments, it also serves to document the algorithm. After proof of concept, in a second step, the code can be optimized, keeping the parts that do not need processing power, and converting only the parts that are computationally intensive. This conversion or optimization can be subdivided in stages, taking advantage of the interactive and experimental nature of Python to define the best solutions, which include, for example, writing modules using Fortran, and in this way, making it possible to concentrate efforts only on certain parts.

4.4. COMMENTS ON OBTAINED F2PY SERIAL AND PARALLEL PERFORMANCE

F2Py, together with F90, were the best performing implementations. F2Py reuses the F90 code and creates a standard Python module, which is then used in the main standard Python code. Despite the overhead added by the wrapper and the Python interpreter, F2Py achieved higher performance for 16, 49, and 81 processes, and also in the sequential implementation. In this case, a Jupyter Notebook extension allows you to compile and import symbols from the Fortran code transparently, making the function available to the Python code, which can then be executed interactively during development. The final code used to run on the execution node, uses the module created by F2Py which is called by a main code in standard Python. Jupyter Notebook is used in every process, including sending to run using the queue manager and scheduler, and reading the resulting output files. As a suggestion for future studies, for 81 processes the performance of F2Py was much better than F90 (1.01 against 1.69 s), and it would be interesting to better understand why this difference occurred.

4.5. COMMENTS ON OBTAINED CYTHON SERIAL AND PARALLEL PERFORMANCE

Cython in the sequential version uses extensions and creates an external module that is called by the main code in standard Python. In the parallel version, the computationally intensive part that uses repetition loops is compiled by Cython, and the rest of the code is executed by the standard Python interpreter, which includes the use of the mpi4py library. The performance of the sequential Cython implementation is between F2Py and Numba, and the parallel is close to F90 and F2Py from 4 to 64 processes. In general, it is a good option when the F90 code does not exist, and we want to use a language that is close to Python and brings with it some advantages such as ease of reading and maintenance, in addition to the fast and interactive development.

4.6. COMMENTS ON OBTAINED NUMBA SERIAL AND PARALLEL PERFORMANCE

Numba uses JIT compilation and the computationally demanding core is placed in a function with an indication for Numba. The rest of the code is executed by the Python interpreter and the parallel implementation uses the mpi4py library. Numba's performance was below F90, F2Py, and Cython, but well above standard Python. The performance of the sequential implementation was worse than sequential Cython, and in the parallel implementation the performance was close to or equal to Cython from 4 to 81 processes. Numba proved to be a good alternative when the F90 code does not exist, or when the Python code exists and we want few changes to the code. In addition, Numba has the advantage of also being able to use the GPU for processing. And since the build is JIT, the code can eventually be optimized during execution making the implementation portable without the need for of a new AOT [2] compilation for each different architecture.

4.7. PROCESSING PERFORMANCE OF T HE NUMBA-GPU IMPLEMENTATION

This section intends to compare the single-node performance of: (i) the single-process Numba-GPU implementation executed in a single-core and one GPU of the Bull B715 or the Bull Sequana-X node; (ii) the single-process F90 serial and parallel implementations executed in one or more processor cores of the Bull B715 or the Bull Sequana-X node, but not any GPU. Please note that the Bull Sequana-X is a much-updated computer node, with newer processors and GPUs, in comparison to the Bull B715 node, being both available in

[2] Ahead-of-time compilation before the execution.

the SDumont supercomputer. It is also worth to note that the F90 implementation serial and parallel performance in the Bull B715 node is equal to the one of the Bull B710 node.

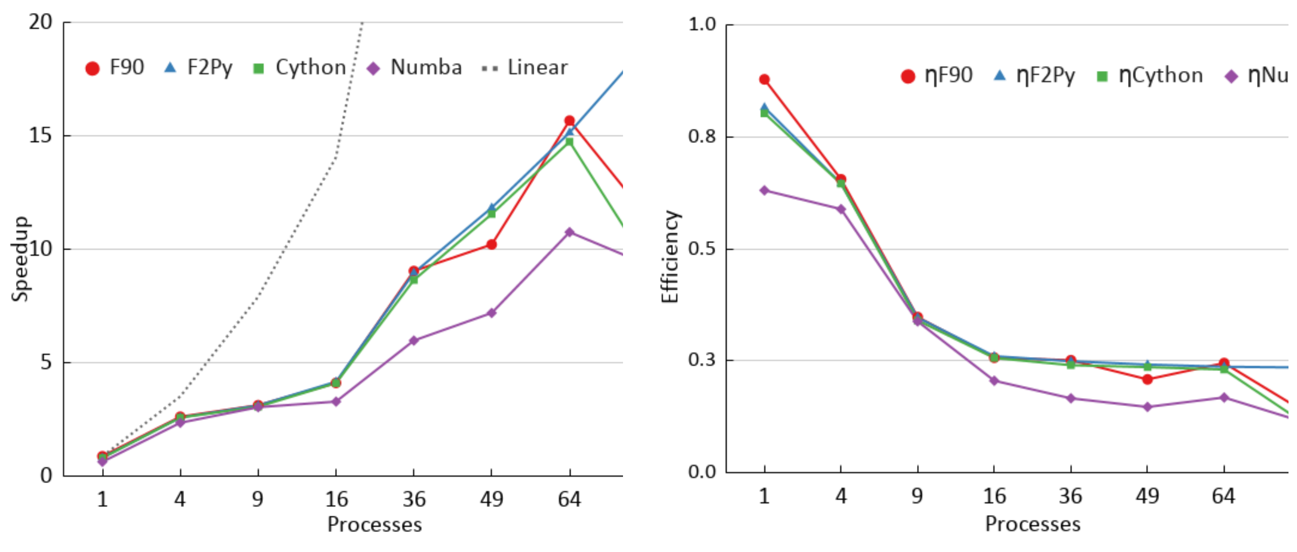
The Numba-GPU implementation was tested in a Bull B715 node using a single core, and the Tesla K40 GPU runs in 22.28 s, close to the execution time of the serial F90 implementation. The same Numba-GPU implementation executed in a Bull Sequana-X node using a single core and a single Volta V100 GPU spent just 8.02 s, slightly more than most parallel implementations executed with 4 MPI processes in the Bull B715 node. The Numba-GPU implementation required more modifications in comparison to the standard serial Python code than the others. Similarly, the calculation of the stencil and consequent update of the grid points compose the main loop of the test case algorithm. Such loop was encapsulated in a function with a Numba hint for execution in a single GPU using just-in-time (JIT) compilation. These execution times correspond to blocks of 8 x 8 threads in the case of the V100 GPU (Bull Sequana-X node) and blocks of 16 x 16 threads in the case of the K40 GPU (Bull B715 node). These block sizes were optimized by trial-and-error.

The single-instruction multiple-threads (SIMT) paradigm models GPU execution, with the problem domain mapped into a grid of blocks composed by a number of threads. Blocks are then split in warps, typically of 32 threads, and executed by one of the streaming multiprocessors that compose the GPU. Numba-GPU uses a wrapper for the C-language CUDA API providing access to most of its resources. The remaining code of the test case algorithm was executed by the Python interpreter, since there is no other computer-intensive part in the code.

The performances of the serial and parallel “pure-CPU” F90 implementation and the single-process Numba-GPU implementation for one Bull B715 node or one Bull Sequana-X node are been taken, and all values refer to the average of 3 executions for each case. The F90 parallel execution in the Bull B715 node was restricted to 16 processes, since the total number of cores of this node is 24. A possible extension of this work would be to employ another level of parallelism, besides the SIMT execution of the GPU, taking advantage of the two processors, each one with 22 cores, and the 4 GPUs of the Bull Sequana-X node. This could be achieved, for instance, running 4 MPI processes in the node, each one employing one of the 4 GPUs. However, to take benefit of a possibly faster multicore and multiGPU implementation would require a more demanding hybrid coding combining the use of MPI processes executed in processor cores calling kernel functions that run in GPUs. Anyway, the performance obtained using the Volta V100 GPU of the Sequana-X shows the processing power of such accelerators, since it is 2.4 times faster than the F90 serial.

Table 1: Performance of the different implementations of the test case, depending on the number of MPI processes: processing times (seconds), speedups and parallel efficiencies. Best values for serial or for each number of MPI processes are highlighted in red. The execution time of the GNU-Fortran-compiled serial code was taken as a reference for the calculation of speedup (highlighted in blue).

	Processing time (s)								
	Serial	**1**	**4**	**9**	**16**	**36**	**49**	**64**	**81**
F90	**19.25**	**21.91**	**7.34**	**6.15**	4.68	**2.13**	1.89	**1.23**	1.69
F2Py	**18.94**	23.60	7.45	6.17	**4.62**	2.15	**1.63**	1.27	**1.01**
Cython	23.97	23.98	7.46	6.29	4.69	2.23	1.67	1.31	2.06
Numba	30.48	30.53	8.18	6.33	5.86	3.22	2.68	1.79	2.07
Python	212.43	227.19	64.74	44.78	33.46	15.21	10.43	7.85	6.70
	Speedup								
F90	**1.00**	**0.88**	**2.62**	**3.13**	4.11	**9.04**	10.21	**15.67**	11.42
F2Py	**1.02**	0.82	2.58	3.12	**4.16**	8.96	**11.83**	15.14	**19.03**
Cython	0.80	0.80	2.58	3.06	4.10	8.64	11.55	14.74	9.36
Numba	0.63	0.63	2.35	3.04	3.29	5.98	7.19	10.75	9.32
Python	0.09	0.08	0.30	0.43	0.58	1.27	1.85	2.45	2.87
	Parallel efficiency								
F90	**1.00**	**0.88**	**0.66**	**0.35**	0.26	**0.25**	0.21	**0.24**	0.14
F2Py	**1.02**	0.82	0.65	0.35	**0.26**	0.25	**0.24**	0.24	**0.23**
Cython	0.80	0.80	0.65	0.34	0.26	0.24	0.24	0.23	0.12
Numba	0.63	0.63	0.59	0.34	0.21	0.17	0.15	0.17	0.12
Python	0.09	0.08	0.07	0.05	0.04	0.04	0.04	0.04	0.04



(a) Parallel speedup. Dashed line: linear speedup. **(b)** Parallel efficiency.

Figure 2: Speedups and efficiencies of the different implementations of the test case, depending on the number of processes. The execution time of the GNU-Fortran-compiled serial code was taken as the reference for speedup calculation.

5. FINAL REMARKS

This work presented a comparison of the most common HPC approaches in Python applied to a given test case and executed in the LNCC Santos Dumont supercomputer. The test case is a 2D heat transfer problem modeled by the Poisson partial-differential equation, which is solved by a finite difference method. It requires the calculation of a 5-point stencil over the discretized domain grid. Their serial and parallel implementations in Fortran 90 were taken as references in order to compare their performance to some serial and parallel implementations of the same algorithm available in the Python environment: F2Py, Cython, Numba, Numba-GPU and the standard Python itself. Processing times, speedups and parallel efficiencies were presented and discussed for these implementations considering a specific problem size of the test case.

The Python environment is a high-level interactive tool for prototyping and rapid development of computer code, allowing the integration of modules written in F90, and the use of a large number of third-party libraries. This work intends to show that Python can also be employed for HPC by means of APIs/libraries like F2Py, Cython, Numba and Numba-GPU. Faster implementations are then generated by porting time-consuming parts of the Python code to a new function that is called from the Python program, except in the case of F2Py, that reuses an existing F90 code. Serial and parallel processing performance results for the given test case show that such approch is feasible. Therefore, Python not only allows the programmer to promptly develop and test the intended Python code, but also to port parts of it in order to attain HPC employing multiple cores and/or GPUs. The resulting code offers the portability of Python, but it provides another level of modularity, since it is possible to exchange a function implemented in Cython, for instance, by another in Numba-GPU according to the available computer architecture.

As previously mentioned, there is a trade-off between languages like F90 or C and the Python environment concerning the easiness of programming and the processing performance. These languages require more programming effort, but facilitate optimization/parallelization. However, availability of HPC resources for Python is increasing.

6. ACKNOWLEDGMENTS

Authors thank LNCC (the Brazilian National Laboratory for Scientific Computing) for grant 205341 AMPEMI (call 2020-I), which allows access to the Santos Dumont supercomputer (node of the SINAPAD, the Brazilian HPC system).

REFERENCES

BALAJI, P. et al. Advanced MPI Programming. In: 2017, **Tutorial at SC17: The International Conference for High Performance Computing, Networking, Storage, and Analysis.**: Argonne National Laboratory, 2017. Available in: <http://www.mcs.anl.gov/~thakur/sc17-mpi-tutorial/>

BEAZLEY, D. M.; LOMDAHL, P. S. Feeding a Large-Scale Physics Application to Python. In: 1997, **Proceedings of the 6th International Python Conference.**: Los Alamos National Lab., USA, 1997. Available in: <http://legacy.python.org/workshops/1997-10/proceedings/beazley.html>

CHEN, C. J. et al. Finite Analytic Method in Flows and Heat Transfer. **Applied Mechanics Reviews**, [S. l.], v. 55, n. 2, p. B34–B34, 2002. Available in: <https://doi.org/10.1115/1.1451170>

DALCÍN, L. et al. MPI for Python: Performance Improvements and MPI-2 Extensions. **Journal of Parallel and Distributed Computing**, [S. l.], v. 68, n. 5, p. 655–662, 2008. Available in: <https://doi.org/10.1016/j.jpdc.2007.09.005>

GROPP, W. et al. A High-performance, Portable Implementation of the MPI Message Passing Interface Standard. **Parallel Computing**, [S. l.], v. 22, n. 6, p. 789–828, 1996. Available in: <https://doi.org/10.1016/0167-8191(96)00024-5>

HINSEN, K. The Molecular Modeling Toolkit: A Case Study of a Large Scientific Application in Python. In: 1997, **Proceedings of the 6th International Python Conference**. [S. l.: s. n.] Available in: <http://legacy.python.org/workshops/1997-10/proceedings/hinsen.html>

IEEE SPECTRUM. **Interactive: The Top Programming Languages 2020**. [S. l.: s. n.]. E-book. Available in: <https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020>

LUNACEK, M.; BRADEN, J.; HAUSER, T. The Scaling of Many-task Computing Approaches in Python on Cluster Supercomputers. In: 2013, 2013 **IEEE International Conference on Cluster Computing (CLUSTER).**: IEEE, 2013. p. 1–8. Available in: <https://doi.org/10.1109/CLUSTER.2013.6702678>

SEHRISH, S. et al. Python and HPC for High Energy Physics Data Analyses. In: 2017, New York, NY, USA. **Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing**. New York, NY, USA: Association for Computing Machinery, 2017. Available in: <https://doi.org/10.1145/3149869.3149877>

SOUZA, C. R. de; PANETTA, J.; STEPHANY, S. Analysis of Communication Performance Using MPI 3.0 Shared Memory Functionality (in Portuguese). **Revista Cereus**, [S. l.], v. 10, n. 2, p. 193–210, 2018. Available in: <https://doi.org/10.18605/2175-7275/cereus.v10n2p193-210>

VIRTANEN, P. et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. **Nature Methods**, [S. l.], v. 17, n. 3, p. 261–272, 2020. Available in: <https://doi.org/10.1038/s41592-019-0686-2>