aa/bb/cc/dd-TDI

COMMON HPC APPROACHES IN PYTHON EVALUATED FOR SCIENTIFIC COMPUTING TEST CASES

Eduardo Furlan Miranda

Master's Dissertation of the Graduate Course in Applied Computing, guided by Dr. Stephan Stephany, approved in MMMM DD, AAAA.

URL of the original document:
<http://urlib.net/xx/yy>

INPE
São José dos Campos
AAAA

PUBLISHED BY:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3945-6923/6921
Fax: (012) 3945-6919
E-mail: pubtc@sid.inpe.br

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA, INOVAÇÕES E COMUNICAÇÕES
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

aa/bb/cc/dd-TDI

COMMON HPC APPROACHES IN PYTHON EVALUATED FOR SCIENTIFIC COMPUTING TEST CASES

Eduardo Furlan Miranda

Master's Dissertation of the Graduate Course in Applied Computing, guided by Dr. Stephan Stephany, approved in MMMM DD, AAAA.

URL of the original document:
<http://urlib.net/xx/yy>

INPE
São José dos Campos
AAAA

Cataloging in Publication Data

Sobrenome, Nomes.

Cutter Common HPC Approaches in Python Evaluated for Scientific Computing Test Cases / Nome Completo do Autor1; Nome Completo do Autor2. – São José dos Campos : INPE, AAAA.

xviii + 121 p. ; (aa/bb/cc/dd-TDI)

Dissertação ou Tese (Mestrado ou Doutorado em Nome do Curso) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, AAAA.

Orientador : José da Silva.

1. Palavra chave. 2. Palavra chave 3. Palavra chave. 4. Palavra chave. 5. Palavra chave I. Título.

CDU 000.000

Informar aqui sobre fontes financiadoras (a modificação desta linha deve ser feita no arquivo publicacao.tex).

ATENÇÃO! A FOLHA DE APROVAÇÃO SERÁ INCLUIDA POSTERIORMENTE.

Mestrado ou Doutorado em Nome do Curso

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to the National Institute for Space Research (INPE) for the resources made available for the development of my research. Also, I am grateful to my advisor, Stephan Stephany, for his guidance and encouragement. Also, I thank to LNCC (National Laboratory for Scientific Computing) for the 205341 AMPEMI scholarship (called 2020-I), which allows access to the Santos Dumont supercomputer (SINAPAD node, Brazilian HPC system).

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. Three test cases are presented: the problem of heat transfer over a finite surface, the discrete Fourier transform of a sequence, and the random forest learning method for classification, regression and other tasks. 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 intends to be a primer for the use of HPC resources in Python.

Keywords: High performance computing. Python language. Scientific computing.

ABORDAGENS COMUNS DE HPC EM PYTHON AVALIADAS PARA CASOS DE TESTE DE COMPUTAÇÃO CIENTÍFICA

RESUMO

Várias das abordagens de computação de alto desempenho mais comuns disponíveis no ambiente de programação Python do supercomputador LNCC Santos Dumont, são comparadas usando um caso de teste específico. Python inclui bibliotecas específicas, ferramentas de desenvolvimento, implementações, documentação e recursos de otimização / paralelização. Ele fornece uma maneira direta de programar em um alto nível de abstração, mas os recursos de paralelização para explorar vários núcleos, processadores ou aceleradores como GPUs são diversos e podem não ser facilmente selecionáveis pelo programador. Este trabalho faz uma comparação de abordagens comuns em Python para aumentar o desempenho da computação. Três casos de teste são apresentados: o problema de transferência de calor sobre uma superfície finita, a transformada discreta de Fourier de uma sequência, e o método de aprendizagem de floresta aleatória para classificação, regressão e outras tarefas. Suas implementações seriais e paralelas no Fortran 90 foram tomadas como referências para comparar seu desempenho com algumas implementações Python seriais e paralelas do mesmo algoritmo. Além dos resultados de desempenho, uma discussão sobre a compensação entre facilidade de programação e desempenho de processamento está incluída. Este trabalho pretende ser uma cartilha para o uso de recursos PAD em Python.

Palavras-chave: Processamento de alto desempenho. Linguagem Python. Computação científica.

LIST OF FIGURES

LIST OF TABLES

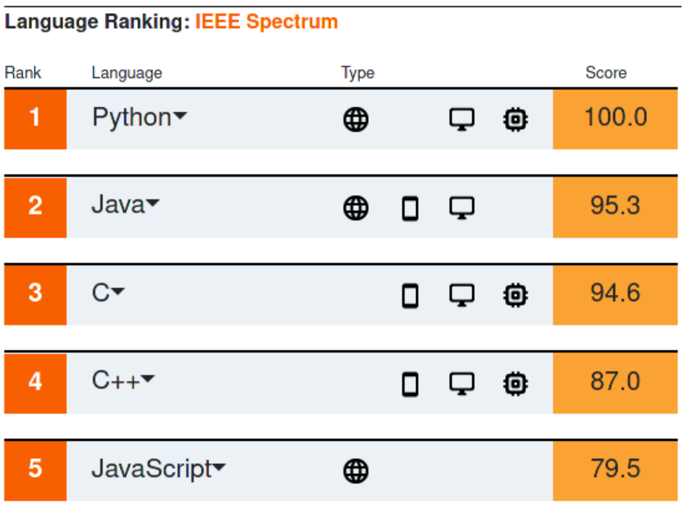
CONTENTS

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 et al., 2013; VIRTANEN et al., 2020a). 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 (section 3.1), allowing fast prototyping. According to the 2019 IEEE Spectrum programming language ranking (IEEE Spectrum, 2020), Python is the most popular, as shown in Figure 1.1.

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 over 80 references for this language (INPE, 2020), including several applications, such as the optimization of a mathematical model to estimate the amount of solar radiation incident on the Earth's surface (SOUZA et al., 2018b), and the use of a neural network for the classification of supernovae (NASCIMENTO et al., 2019). Python availability reaches compiler packages like the Intel one (CIELO et al., 2019) or most

Figure 1.1 - Ranking of most popular programming languages, according to the *IEEE Spectrum ranking*.



Language Ranking: IEEE Spectrum

Rank	Language	Type	Score
1	Python	🌐 🖥 ⚙	100.0
2	Java	🌐 📱 🖥	95.3
3	C	📱 🖥 ⚙	94.6
4	C++	📱 🖥 ⚙	87.0
5	JavaScript	🌐	79.5

Source: Adapted from IEEE Spectrum (2020).

supercomputer programming environments. Application programs implemented in languages like Fortran 90 or C, even demanding massive parallel processing, can be encapsulated in the Python environment by means of wrappers in a modular way. Such flexibility facilitates 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).

1.1 Objectives

This work aims to explore the most common parallelization approaches available in the Python ecosystem, which includes libraries, frameworks, tools, etc. It intends to describe and discuss some High Performance Computing (HPC) approaches that are available in the Python environment, such as parallel processing, or general purpose graphics 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; BARNEY, 2021)). Some general considerations about this work follows in order to explain its scope in the vast Python environment:

a) 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;

b) Python implementations of this work include HPC solutions for standard Python, Cython, Numba, Numba-GPU and F2PY, but there are many others;

c) 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;

d) 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.

As part of the objectives of the work, two articles were submitted, approved and published. One of them was presented at a national congress, the XV Brazilian e-Science Workshop (BreSci), an event that is part of the XLI Brazilian Congress of Informatics Society (CSBC-2021) (MIRANDA; STEPHANY, 2021b), and the other article was published in the journal Cereus Magazine (MIRANDA; STEPHANY, 2021a). They are two different articles and both cover the stencil case study only, mainly due to limited text space available for publication.

1.2 Text organization

The following is a brief description of the remaining chapters of this document, showing its organization:

- chapter 2: Environment resources, showing the main characteristics and resources of the environment.

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

- chapter 3: HPC approaches to Python programming, describing each of the main approaches, and summarizing the others.
- chapter 4: Case studies, comparison of implementations in Python with implementations in F90, referring to selected algorithms.
- chapter 5: Parallel performance analysis in Python, comparing the computational performance of the selected HPC approaches for the Python environment.
- chapter 6: Final remarks.

2 PYTHON ENVIRONMENTAL RESOURCES

Python is a high-level, general-purpose, interactive language, with dynamic typing and automatic memory management, supporting the imperative, functional, and object-oriented programming paradigms, and in combination with its libraries allows two ways of execution, interpreted and compiled. Among the supported paradigms, the functional with division of processing in independent tasks allows a more efficient parallelization. It is one of the most widely used languages, easy to learn, easy to read and maintain, it is portable across platforms, extensible, scalable, and has support, among others, for databases, and GUI programming (SANNER, 1999).

2.1 Programming paradigms

There are several taxonomies proposed for programming paradigms; one can try to summarize, in general, imperative, declarative and object-oriented. The imperative paradigm consists of a sequence of commands that has an explicit implementation, that is, explaining *how to do it* to execute an algorithm. In this paradigm, the source code, for example in C or F90, is usually compiled by generating code in machine language, which is subsequently executed. Generally, in these languages, the imperative programming paradigm is used with compilation ahead-of-time (AOT). The declarative programming paradigm, on the other hand, enumerates tasks to be performed, leaving its implementation implicit, that is, showing what the *algorithm should do*. The declarative paradigm can be subdivided into functional, logical programming, or directed to a database (SEBESTA et al., 2016).

The Python language allows you to write code following a functional declarative programming paradigm when in combination with your standard library or with external libraries. This paradigm is based on the application of functions to data passed as arguments, which allows the interpreter to generate the intermediate representation composed of independent tasks corresponding to each function call, which simplify the parallelization (SINGH, 2010).

2.2 Compiled and interpreted languages

In addition to the taxonomy of programming paradigms, programming languages can be divided and grouped according to the way they are executed or implemented, in two large groups: compiled languages and interpreted languages, with the first group generally corresponding to imperative programming, but not necessarily from the second group with declarative programming. For example, Python and C++, depending on the implementation employed, may fall into one group or another. The compiled languages imply the compilation of the entire program, generating object code in machine language and requiring memory for this, but allowing a quick execution thanks to the optimizations of the compiler and not requiring new compilation in repeated executions in the same architecture. On the other hand, the interpreted languages execute instruction by instruction, without generating native object code, requiring less memory, but with slower execution and possibly requiring a new compilation for an intermediate representation at each repetition. It should be noted that Python can reuse the generated intermediate format, avoiding the recompilation of already compiled code. In addition, while machine code generation does not require less memory, the full programming environment requires a virtual machine that requires a lot of memory. The use of less memory is even more evident for extensive Python code, despite the memory required by the Python virtual machine. Note, however, that these definitions are general, as it is possible to implement interpreters for compiled languages, as well as compilers for interpreted languages. A good example of this is the C++ interpreter from CERN (VASILEV et al., 2012).

Python, because the standard implementation is an interpreted language, requires an interpreter which is a program that executes instruction-by-instruction Python code or in instruction blocks grouped in a script. An interpreter usually translates the original language into an intermediate representation, which is performed by the associated virtual machine. It is possible to have different virtual machines for the same interpreter, each associated with a specific processor architecture. There are several interpreters available, which may or may not support all the features of the Python language, and the reference implementation, CPython, translates the source code of the language into an efficient intermediate representation (bytecode), which is then executed using a Python virtual machine. In the case of libraries used with Python, the interpreter accesses the functions or routines of these libraries in intermediate representation to be executed by the virtual machine, or less frequently, in machine language, to be executed directly by the processor (SANNER, 1999).

More recently, the so-called runtime compilation, or just-in-time (JIT), was introduced, in which the interpreter compiles the intermediate representation at runtime, generating code in machine language for execution in the processor. Thus, at runtime, the identification of the types of variables is done, allowing the appropriate optimizations. So, the use of Python with libraries that allow JIT compilation combines the advantages of both languages, that is, the freer and clearer syntax of the interpreted languages with the better performance of the compiled languages. An example of a

JIT interpreter for Python is Pypy (BIHAM; SEBERRY, 2006).

2.3 Python intermediate representation

When we execute Python code, the Python reference implementation, CPython, first compiles the source code to an intermediate representation (called bytecode), and then, in a second step, executes it in an interpreted way. After compilation, which is done transparently, the bytecode is then sent to be executed by a Python Virtual Machine (PVM), which is part of the Python system, and which would be the interpreter itself. The intermediate representation has the advantage of being platform independent, and once the bytecode is generated, it can be executed without changes on a different platform that has a specific PVM for the target machine. The intermediate representation is generated through the analysis of the Python source code, and some optimizations are made, however many of the language analyzes, such as type checking and other characteristics of dynamic languages, are done at runtime by the interpreter. Depending on the resources used, bytecode is stored in an external file, and can be reused in later executions, saving compilation time. In this case, if the source code does not change, there is no longer a need for the compilation step, and for execution bytecode and the virtual machine are enough. All of these steps are automatically managed by Python and are transparent to the user. Python bytecode is also used by other system components, as is the case with the Numba compiler, which uses it during runtime to compile certain machine code snippets into machine code, making hybrid execution, part interpreted using bytecode, and part executing native code (LAM et al., 2015).

2.4 Python decorators

Python treats classes and functions as objects, allowing you to pass classes or functions as an argument to other functions. The Python language incorporated in its more recent versions the use of the so-called decorators, which improves the syntax for passing a function as an argument of another function, without explicitly modifying it, but extending its behavior. Also, a function can return another function. As an example, decorators, combined with the JIT Numba compiler, allow you to define functions that should be compiled only at run time. In this case, the interpreter, instead of using its representation of the function, uses the machine language code generated by Numba. Thus, the Python user has the impression of executing commands in an interpreted language, but taking advantage of the performance obtained by compiling just-in-time (LAM et al., 2015).

2.5 Python libraries

Python's flexibility allows the user to choose from several parallelization and optimization features through libraries or different implementations in imperative languages like C or F90. As an example, the library for scientific computing NumPy, which uses code implemented in C/C++ and F90 (WALT et al., 2011), has a tool called F2PY that generates an F90 program interface for Python (PETERSON, 2009). There are also compilers and libraries, such as Cython, that generate C/C++ code to be statically compiled (BEHNEL et al., 2010). Another example is the Numba library, which allows compiling just-in-time for a subset of functions in Python and NumPy (LAM et al., 2015). `mpi4py` is a library, as its name suggests, that allows parallelization using the Message Passing Interface (MPI) standard, for execution using multiple processes (DALCÍN et al., 2008). On the other hand, Dask is a library for parallel processing that integrates with specific libraries that allow more efficient parallelization by dividing processing into independent tasks (ROCKLIN, 2015). In addition to these, there are several other libraries and HPC solutions for Python, such as libraries related to SMD, Cluster, Cloud, Grid, and Distributed Computing (PALACH, 2014). Another example of this is PyTorch, which is an open source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing. PyTorch has support for parallelization on machines with multi-core processors and with GPU (KETKAR, 2017).

In short, the Python user has access to more than sixty libraries and solutions related to parallel processing (PALACH, 2014). The Anaconda open-source distribution, focused on scientific computing, which aims to simplify package management and deployment, has a cloud-based repository with more than seven thousand packages for data science and machine learning. Anaconda is available for use in SDumont. The current version of Python, as of December 2020, is 3.9.1. The software composed of the Python language and its standard library, the external libraries, and the interpreter and virtual machine used, constitutes the so-called Python programming environment (or Python ecosystem).

Throughout this text, for the sake of simplicity, each core of a multi-core processor will be denoted as CPU, and each general-purpose graphics accelerator card (GPGPU - General Purpose Graphics Processing Unit) will be denoted as GPU. In addition, High Performance Computing, which obviously includes parallel processing on CPUs and / or GPUs, will be denoted as HPC.

2.6 Jupyter Notebook

The Jupyter Notebook, with the new Notebook interface, JupyterLab, is an interactive development environment that has the flexibility to allow you to combine executable code snippets to solve a problem, with explanatory text, and calculation results including graphs, in addition to having an interactive authoring application running in the web browser. In this way, it generates a powerful work environment for rapid development and prototyping, promoting integration between the various components of this environment, allowing easy visualization and analysis, and generating multimedia web documentation (PERKEL, 2018). It has a server-client architecture that allows, for example, to run the notebook server on a remote machine (a cluster, or else a supercomputer) and interactively develop a prototype on a local laptop, while the execution is done transparently on the remote machine. The development interface in the web browser is standardized and is the same regardless of where the notebook server is running, making it easy to use in different environments and systems. It is even possible to use several notebook servers at the same time, each running on different machines, and being viewed in different windows or tabs of the web browser on the local machine, making it possible to develop and run code on different machines at the same time. Depending on the configuration, it is possible to use or combine several programming languages, for example it is possible to develop code in F90 or C. It is also possible to access the operating system shell and perform most of the tasks and features from the command line. It also creates a stand-alone document on the disk containing what has been made or changed, which can be viewed in an appropriate document reader or used in another notebook session. When used in conjunction with Conda (GRÜNING et al., 2018), an open source environment and package management system, it provides a quick and easy environment and system to install, run and update Python packages and resolve their dependencies. It is also possible to work with different Python environments (including JupyterLab), which can be easily created, saved, loaded and switched. Conda also has features to list current packages and install on another machine to allow reproducibility. Conda allows you to use different versions, for example of Python or JupyterLab, in completely separate environments and switch between them. It is also possible to create stacked environments, to allow, for example, adding JupyterLab to an existing environment, without modifying it. This feature can be useful, for example on SDumont, if the existing Python distribution does not have the JupyterLab or other package. Conda also has a large online package repository (BISONG, 2019).

2.7 Related work

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

3 PERFORMANCE APPROACHES FOR PYTHON

The purpose of this chapter is to briefly describe the main parallel approaches to Python coding, as well as some other features used in the implementations described in this work. In this text, the term parallel approach is used instead of HPC, referring to the simultaneous execution of several processes or threads, the latter in CPU (central processing unit, or processor core) or in GPU (graphic processing unit) . The intention is not to be as complete and comprehensive as possible, but rather to provide some support, for example, for those just starting to use the Python environment on the SDumont supercomputer. Note that while several approaches are described, only a few have been selected for use in the chapter 4 case studies, and most of them are continually being developed and improved.

3.1 IPython Parallel

IPython alone is a command shell for interactive computing in multiple programming languages, originally developed for Python, with a number of features typical of others shells, but allowing for interactive execution of tasks. It is complemented by IPython Parallel (ipyparallel), which constitutes an abstraction layer that provides support for interactive parallel processing. IPtyhon Parallel allows you to configure a parallel execution environment with the desired architecture, allowing applications to be developed, executed, monitored and debugged interactively, supporting various types of parallelism such as data parallelism, applicable to arrays. It allows to emulate SPMD (single program, multiple data) or MPMD (multiple program, multiple data) architectures, run parallel programs with the MPI message exchange library, perform *task-farming*, in addition to enabling personalized approaches. Usage examples include fast, interactive parallelization with just two lines of code, *embarrassingly parallel* algorithms (trivially paralyzed because they are not data dependent); use an IPython session on a laptop to trigger the execution of an MPI program on a supercomputer, interactively analyze large distributed data sets using *matplotlib* for data visualization, interactively develop parallel algorithms, test and debug them them; generate a large distributed parallel system through the union of multiple MPI programs executed in different systems or use dynamic load balancing in the execution of a set of tasks using several processors of a shared memory machine. Thus, the user can run parallel programs that involve input / output of a large volume of data and analyze the results (FARRELL et al., 2018).

3.2 MPI for Python

MPI for Python (`mpi4py`) is a module that allows Python code execution using the MPI message exchange library (Message Passing Interface) for parallel processing. It allows the parallel execution of a Python script when triggering multiple MPI processes on one or more nodes of shared memory, each with multi-core processors. Currently, `mpi4py` supports the MPI-3 standard, although its API is compatible with that of the MPI-2 standard for the C++ language. Thus, it makes the parallel execution of Python scripts accessible, providing most of the MPI functionality and providing communication between processes of Python objects such as NumPy arrays.

In the case of the communication of Python objects not supported by the MPI standard, you can use the Python *Pickle* module, which generates a sequence of bytes from the object to, after the communication, return the original object (the object is *pickled* resulting in a string to later be *unpickled*). Original MPI functions, are called with a capital letter (example, `MPI_Send`), while the functions that support the communication of these sequences are called with a lower case letter (example, `MPI_Send`). Parallel input and output, according to the MPI-2 standard, are also available, requiring a parallel file system (DALCÍN et al., 2008; GROPP; LUSK, 1996).

3.3 Cython

Cython is a compiler for the Python language, and for its own Cython extensions, which allows you to generate code in the C language automatically from Python instructions using a static optimizing compiler to obtain better computational performance (BEHNEL et al., 2010). There are specific Cython extensions for the Python language, so you have your own Cython language, based on Python itself. Thus, Cython not only has interfaces for the same Python libraries, as with NumPy (WALT et al., 2011), but it also allows interfacing with existing code or with libraries written in C/C++. Cython allows you to combine Python's fast development environment with the performance of optimized programs compiled in C, instead of using the standard Python library (`libpython`), supporting all language, and variable granularity, to make it simpler the task of translating to C avoiding using the internal Python library (`libpython`), to achieve greater performance. Cython is typically used to create Python extension modules, from annotated code similar to Python, which is compiled for C/C++ and automatically " encapsulated " (wrapped) in an interface in Python syntax, in code interface, produced a standard extension module, which is then used in a common Python program. The Python code is translated by the

Python interpreter into the intermediate representation bytecode, and from there, the CPython interpreter generates C/C++ code, which is compiled and results in an executable program in machine language. Cython uses the Python annotations feature (*annotations*, function argument metadata). The performance of Cython depends on the type of C code generated in the Python compilation, and the way the machine code is generated by the C compiler (BEHNEL et al., 2010).

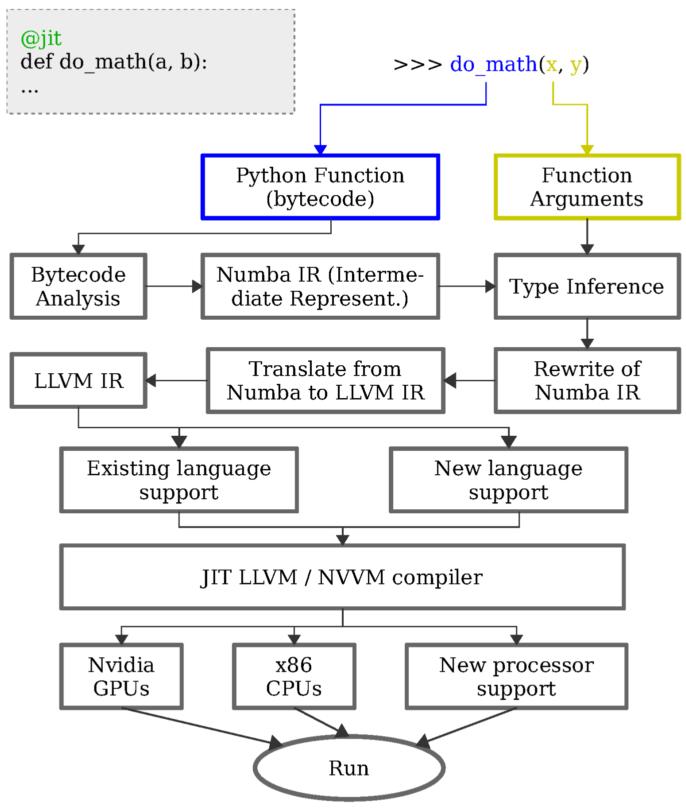
3.4 Numba

Numba is a just-in-time compiler that converts a subset of Python and NumPy library functions into optimized machine code using the LLVM (subsection 4.1.6) (LATTNER; ADVE, 2004). Numba is available in the Python (MAROWKA, 2018a) Anaconda distribution. NumPy, seen in section 3.7, is a library that supports large multidimensional arrays and provides mathematical functions that can be applied with these arrays as an argument (WALT et al., 2011). The LLVM package allows generation of the optimized machine code (for example, with vectorization) and its execution in multi-core processors or in GPU, either in a shared memory machine, or in a cluster or in a supercomputer (LAM et al., 2015). LLVM currently supports compilation of languages such as Ada, C/C++, D, Delphi, F90, Haskell, Julia, Objective-C, Rust, Swift, among others. It is based on converting the code to its own intermediate representation (IR - Intermediate Representation), which is strongly typed and follows the RISC standard (Reduced Instruction Set Computing). Thus, Numba allows for rapid code generation, with generally only minor changes to the original Python code. The Figure 3.1 shows the diagram representing the phases of interpretation and JIT compilation of Numba.

3.5 PyTorch

PyTorch is an open source machine learning library based on the Torch library, which provides resources for parallel execution on multi-core processors or GPU (KETKAR, 2017). Torch is an open source library for machine learning that provides a programming environment for scientific computing and uses the Lua programming language, from script, in its compiled implementation just-in-time, which is based on the C language. It is important to note that Torch supports operations with multidimensional arrays and tensors. Although the development of Torch has been stalled since 2018, PyTorch continues to be developed and updated, being used for applications such as computer vision and natural language processing. PyTorch has interfaces for Python and C++ languages and provides scalable and distributed training, optimizing development, features supported by Torch (COLLOBERT et al.,

Figure 3.1 - Diagram of Numba JIT interpretation and compilation phases.



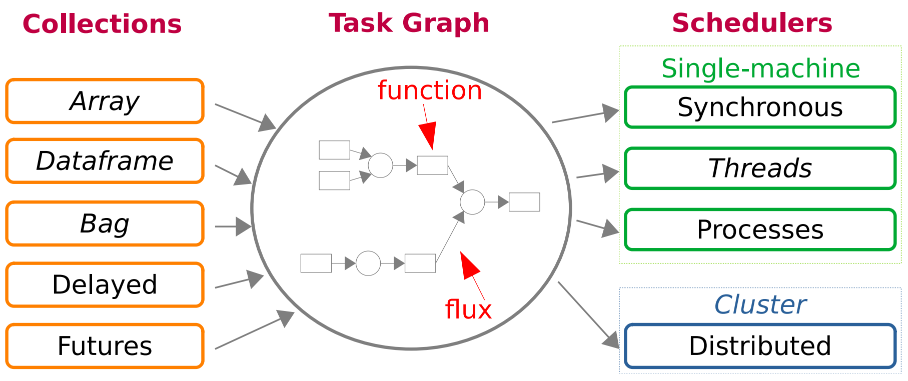
Source: Adapted from Lam and Seibert (2019).

2002). PyTorch's main developer is Facebook, which uses mainly for translations and natural language processing (NLP), processing approximately 6 million translations per day, which include distributed and multitasking training for multiple complex models at the same time (KETKAR, 2017) . PyTorch allows immediate execution of calculations with tensors by means of computational graphs generated dynamically, and in each, the vertices correspond to the tasks to be performed and the edges define the order of execution of these tasks. Each task can be performed independently, using parallel processing, for example with GPU (FEY; LENSSEN, 2019).

3.6 Dask for Python

Dask is a programming environment that optimizes the parallel execution of Python programs. Dask integrates with other libraries such as NumPy, Pandas, and Scikit Learn, and has a scheduler (scheduler) capable of providing parallelization in the execution of programs, whether on a simple laptop multi-core, cluster, or even on

Figure 3.2 - Diagram of the task graph and Dask schedulers.



Source: Adapted from http://dask.org/

a supercomputer. Dask integrates well with the Python environment, requiring few code changes. In addition, it provides the automation of parallelization by dividing the processing into independent tasks and scheduling them for execution according to the available computational resources (shared memory node, processors, cores, GPU, etc.). Dask is installed automatically when installing the Anaconda distribution, which is the most popular of the Python versions for Data Science, thus providing efficient parallelization. Some notable features of Dask are the support for multidimensional arrays, the low level interface to allow optimizations, and the full integration with Python. Perhaps the main feature of Dask is its declarative programming, resembling Python. The Dask scheduler is based on the generation of a graph, in which the nodes represent Python functions and the lines, the flow of Python objects between the nodes, as illustrated in Figure 3.2. This allows you to identify independent tasks, which correspond to the functions of each node, and which allow you to efficiently process large collections of data. After the graph is generated, the Dask task scheduler distributes them for execution, with two options: the *single machine scheduler*, which is simpler and applies to a local shared memory machine, and the *distributed scheduler*, more complex, as it applies to a distributed memory machine, such as a cluster (ROCKLIN, 2015; ROCKLIN, 2021).

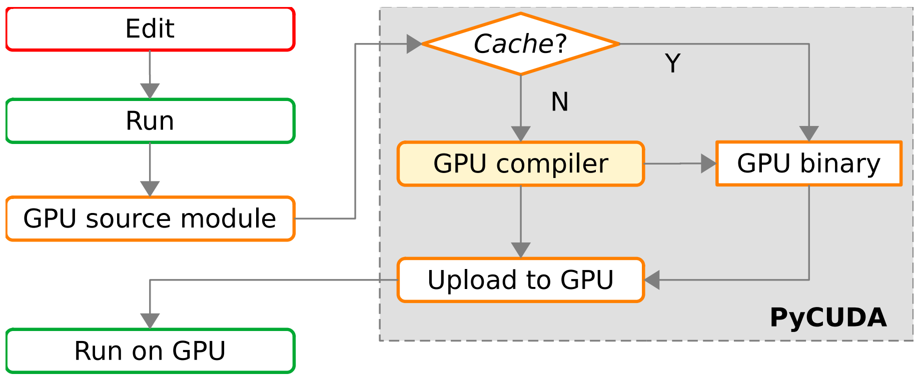
3.7 NumPy

The NumPy library is the basic library for scientific computing in Python, allowing the use of multidimensional arrays, definition of arbitrary data types, integration with databases, and containing communication functions, functions for linear algebra, Fourier transforms , functions for random numbers, as well as tools to integrate C/C++ or F90 code (WALT et al., 2011). Regardless of its use in scientific computing, NumPy serves efficiently to process multidimensional data in general. In particular, Intel's NumPy implementation allows, for certain operations, vectoring by means of SIMD instructions (single instruction, multiple data) to run on Intel processors with AVX vector instructions, in addition to the use of multi-threading to run on multiple cores of program loops, also on a range of Intel processors. The Python language, at the beginning of its development, was not designed for numerical processing (number crunching), but its characteristics led to the development of several program libraries, in particular, the NumPy library. Conversely, this has led to improvements in Python syntax, such as handling arrays indexing. Some characteristics of NumPy can be compared to those of MATLAB (software for numerical calculation), such as interactivity, rapid development, and operations with arrays and matrices. Several other libraries rely on and work in conjunction with NumPy, such as OpenCV (computer vision) that uses it to store and manipulate data. In NumPy, arrays can be pointed to memory regions allocated by extensions in C/C++ or F90, without the need for copying, thus allowing a certain degree of compatibility with existing numerical libraries, such as BLAS and LAPACK (libraries for algebra) linear). One of the limitations of NumPy is the algorithms that are not conducive to vectorization, which makes them slower to be processed by the CPython interpreter, which is the reference implementation of the Python interpreter. CPython was written in C and compiles the Python code for the intermediate representation bytecode (WALT et al., 2011).

3.8 F2PY

F2PY (F90 for Python) is part of the NumPy library, and allows the use of external Python functions compiled with F90 or C, which offer good computational performance. F2PY wraps calls to these functions in interfaces that follow Python syntax and can therefore be imported into Python code. F2PY was designed to allow reusing existing F90 code, with few changes, and thus to combine the ease of Python development with the performance of F90. From an existing F90 code, F2PY generates a Python interface for a new extension module accessible to the Python user. If the F90 code does not exist, it is possible, for example, to select the computationally intensive part of the Python code, and rewrite it using F90 to increase performance. F2PY has good integration with Python and allows F90 code in a Jupyter Notebook cell, transparently creating modules to be used in Python code. It is also possible to use F2PY on the command line to generate modules

Figure 3.3 - PyCuda JIT interpretation and compilation diagram.



Source: Adapted from Klöckner et al. (2012a)

independent of the Jupyter environment (PETERSON, 2009).

3.9 PyCuda

PyCuda is a Python library and programming environment, which works as an interface to the CUDA parallel programming API, written in C++, with features and resources such as comprehensive implementation, guarantee of resources between class initialization and completion, dependency control for allocated memory control, abstractions to facilitate CUDA programming, automatic translation of CUDA errors for Python exceptions, JIT compilation using LNCC / NVCC, has little or no overhead from wrapping, comes with library optimized for GPU for linear algebra, reduction, search, and additional packages for FFT (fast Fourier transform) and LAPACK, in addition to complete documentation. PyCuda also supports threading with different contexts for each thread, albeit with limitations on the freeing of dynamically allocated memory. It is also possible to use the CUDA-GDB debugging tool to debug Python / PyCuda scripts, and the CUDA profiling tool to view information such as routines or functions and tree runtimes of calls that shows which ones called or were called by others, either to run on CPU or GPU (KLÖCKNER et al., 2012b). The Figure 3.3 shows the JIT PyCuda interpretation and compilation diagram, where we have an interpreted phase of tests and execution on the CPU, and a compilation phase for the GPU that uses a temporary storage of binary codes, to avoid repeating compilation at each run.

3.10 CuPy

It is a library compatible with the NumPy library and which uses the CUDA language to allow execution on Nvidia GPUs. It is based on other libraries developed for processing on these GPUs, such as cuBLAS, cuDNN, cuRand, cuSolver, cuSPARSE, cuFFT and NCCL. In some cases, speedup in relation to CPU execution reaches 100. CuPy, given its good compatibility with NumPy, can replace it in most cases. CuPy supports, among other features, methods, indexing, *broadcasting*, and easy encoding of kernel Custom tail (NISHINO; LOOMIS, 2017).

3.11 mpi4py-fft

mpi4py-fft is a Python package to calculate Fast Fourier Transforms (FFTs), which internally uses Message Passing Interface (MPI) to Python (mpi4py) to calculate large arrays, which are distributed and communications are handled internally using mpi4py. A generic algorithm is used for distributing large arrays, which allows distributing any set of multidimensional array indices. The distribution can be slab decomposition, pencil decomposition or other for arrays with a large amount of dimensions. The library has an interface to the FFTW serial library (FRIGO; JOHNSON, 1998), that can be used like the pyfftw library, and in order to work, an MPI library must be installed and configured in the operating system, as well as the FFTW library. Conda, an open source environment and package management system, can be used to install dependencies, making mpi4py-fft easier to use. (MORTENSEN et al., 2019)

3.12 pyFFTW

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris. (GOMERSALL, 2021)

3.13 PARF

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris. (BOULESTEIX et al., 2012)

3.14 Pandas

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris. (MCKINNEY et al., 2011)

3.15 Scikit-learn

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris. (PEDREGOSA et al., 2011)

3.16 SciPy

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu

libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris. (VIRTANEN et al., 2020b)

3.17 Joblib

Library and set of tools for light channeling (chained processing), focused on on-demand computing and transparent parallelization, especially for large arrays NumPy. Among the features are preservation of Python function outputs, and a multiprocessing system for repetition loops (FAOUZI; JANATI, 2020).

3.18 Other solutions

The following is a summary of some other solutions for Python that are being actively developed, and that offer parallel processing, multiprocessing, or interfacing capabilities with C or F90. It does not include cloud computing, in grid, and related projects, such as distributed file systems.

3.18.1 Intel Python distribution

Set of Python packages and libraries optimized for Intel processor architectures, in order to optimize the performance of scientific computing and data science applications. It includes specific optimizations for Intel processors, typically vectoring instructions, multithreading, and Intel libraries aimed at optimizing the processing of packages such as NumPy, SciPy, and Scikit-learn. These Intel libraries are included in distributions such as Numba and Cython. Its suite includes the Intel Math Kernel

Library, Intel MPI, Intel Tread Building Blocks, and Intel Data Analytics Acceleration Library (CIELO et al., 2019).

3.18.2 Python and Nvidia GPU

Nvidia's GPUs are used in several HPC (HOLM et al., 2020) projects and are also present in 100 systems on the TOP500 list (ranking of the 500 supercomputers). A simple search for the word "Python" on Nvidia's site, returns 5,230 results. Python is used in the Nvidia DIGITS environment for *deep learning*, composed of the cuBLAS, cuDNN, NCCL, NVCaffe, Torch, and TensorFlow (YEAGER et al., 2015) libraries. It is also used in the Nvidia DALI library (*Data Loading Library*) to speed up data preprocessing for *deep learning* (GAYER et al., 2019) applications. Another library is Nvidia Theano, for definition, optimization, and analysis of mathematical expressions involving multidimensional arrays (BERGSTRA et al., 2010). The Python TensorFlow module is a machine learning library, which runs on CPUs and GPUs (BROWNLEE, 2016). Nvidia also contributes to the design of the LLVM compiler, allowing, for example, the F90 LLVM compiler to generate code for GPU (OSMIALOWSKI, 2017). Using Python and Numba is one of the easiest ways to use GPU for computing (ODEN, 2020).

3.18.3 FEniCS

Computing platform for solving partial differential equations. It has resources to work with finite elements efficiently. Scalable for high-performance clusters is designed for parallel processing, and allows rapid prototyping, in addition to scaling the same code for HPC. As an example, a thermomechanical simulation can be initially developed on a *desktop* computer, and then the same code can run on a larger scale using 24,000 parallel processes (ALNÆS et al., 2015).

3.18.4 dispy

Structure for creating and using clusters for parallel and distributed computing, including multiprocessing, and processing in cluster, grid, and cloud. It fits well in the paradigm of parallel data processing, and has resources for communication between tasks, client and server modules, and scheduler for shared execution. It is a generic and comprehensive environment for creating and using clusters to perform parallel computing between multiple processors. It has features for data parallelization, and communication of tasks in a concurrent, asynchronous, or distributed manner (AGIUS; INGUANEZ, 2019).

3.18.5 Parallel Python

Module that provides code parallelization mechanisms in multiprocessor systems or in clusters, using processes and inter-process communication. Features include dynamic allocation of processes and resources at runtime, load balancing, fault tolerance, and cross-platform operation (PALACH, 2014).

3.18.6 Ray

System for building and running distributed applications, supporting GPU, multiprocessing, and cluster execution. It includes libraries to speed up machine learning, and can be used in conjunction with libraries like PyTorch, TensorFlow, Keras, and others (MORITZ et al., 2018).

3.18.7 Celery

Environment for distributed applications, consisting of asynchronous (background) or synchronous (waiting until they are ready) task execution queues, based on parameter passing, with support for scheduled or real-time execution, written in Python. Tasks can be executed concurrently on one or more execution nodes (CPUs), using multiprocessing, or co-routines. Celery is used, for example, in services like Instagram to process millions of tasks (MCLEOD, 2015).

3.18.8 Charm4py

Environment for distributed computing and parallel programming, built on top of Charm ++ which is a programming language and environment for parallel programming supported by an adaptive execution environment. Charm4py supports migrable objects and calling remote methods asynchronously (CHOI et al., 2021).

3.18.9 Deap

Computational environment for rapid prototyping and testing of ideas. It also works with multiprocessing mechanisms and SCOOP (seen in the following item). It allows the creation of new types, customization of initializers, intelligent choice of operators, and writing of algorithms (De Rainville et al., 2012).

3.18.10 SCOOP

Module for distributed tasks, allowing concurrent parallel programming in various environments, from heterogeneous *grids* to supercomputers. Features include working with multiple computers on a network, creating multiple tasks within a task, easily paralleling serial code, and efficient load balancing (HOLD-GEOFFROY et al., 2014).

3.18.11 Pyro

Library that allows you to easily build applications where objects on a network can communicate with each other. Common Python methods can be used for calling objects on other machines, including call and return parameters. Pyro is in charge of locating the object and the machine, to execute it (BLANK et al., 2003).

3.18.12 PySpark

High-level library and interface and engine optimized for the Spark environment, which is a unified analysis engine for large-scale data processing. It offers more than 80 high-level operators for building parallel applications. It consists of libraries that include a database interface, machine learning, and graphics production. It also allows you to work with systems like Hadoop (distributed computing) and cloud computing, among others (DRABAS; LEE, 2017).

3.18.13 PyPy

Alternative Python implementation that uses JIT crawled compilation, and is compatible with the Python reference implementation, with the exception of its *extensions*. It has tracking of frequently performed operations, in order to compile for machine code only those parts of the code that require greater processing capacity, making the execution mixed, one part interpreted and the other native (BIHAM; SEBERRY, 2006).

3.18.14 Python Multiprocessing

The *multiprocessing* module, which is part of the standard Python library, allows parallelism for the concurrent execution of processes, executed in the multiple cores of the processors of a shared memory machine. Thus, it cannot be used on a distributed memory machine, such as a cluster or a supercomputer. It is accessed through its API that allows you to easily use the data parallelism paradigm (SINGH et al., 2013).

3.18.15 Python Threading

Module *threading*, which is part of the standard Python library, allows parallelism for the concurrent execution of threads executed on the multiple processor cores of a shared memory machine, with all threads are part of a single process and everyone has access to the memory of that process (MAROWKA, 2018b).

3.18.16 Pythran

Similar to Cython, Pythran is a *ahead of time* compiler for a subset of the Python language, with a focus on scientific computing (like Numba), which takes a Python module containing *annotations* and some interface descriptions, and generate a native module from optimized high-level constructions, requiring only *annotations* of type for exported functions. You can also generate calls to OpenMP (library for shared memory multi-process programming), and use SIMD instructions (GUELTON, 2018).

3.18.17 Nutika

Another project similar to Cython, with a focus on compatibility and simplicity, which supports all Python language, does not require *annotations*, and uses calls to the *libpython* library. It's a compiler written in Python that compiles Python source code to C source code, applying some compile-time optimizations in the process, compatible with all Python libraries and modules and has an interface for programs compiled in C (Van Rossum et al., 2007).

3.18.18 Pyrex

Project with some similarity to Cython, with a focus on helping to write Python extension modules, making it easy to create code required to interface modules, using a language with syntax similar to Python, which allows writing code that mixes Python and data types C, and compile in an extension module for Python (OLIPHANT, 2007).

3.18.19 Boost.MPI

It provides *links* (*bindings*) Python built on top of the C++ library *Boost.MPI*, which is a C++ interface for MPI, through the *Boost.Python* library, as a alternative interface for MPI. Using Python with *Boost.MPI* has some advantages, and as examples, we can mention the Python fast development environment, and also simplifications in the code, because using *Boost.Python* it is not necessary to write the initialization code , as in C++, as this is already done automatically when loading the *Boost.MPI* module in Python code. To transmit Python objects, it is possible to do so in several ways, for example, using *pickling* (preparing sequentially, suitable for transmission) in the sending process, and then *unpickling* in the receiving process (ABRAHAMS; GROSSE-KUNSTLEVE, 2003).

3.18.20 PyOpenCL

It works as wrapper for the OpenCL [1] library API to allow Python to access it. OpenCL was developed in C++ and allows you to perform tasks in parallel on multi-core processors and on different processing accelerators, such as GPGPU (General-Purpose Graphics Processing Units), DSP (Digital Signal Processors), FPGA (Field Programmable Gate Array), and others. OpenCL was initially developed by Apple, and collaborates with AMD, IBM, Qualcomm, Intel, and Nvidia, who seek to provide a common API to the different existing processing accelerators. Some implementations of OpenCL use the LLVM compiler (KLÖCKNER et al., 2012b).

[1]OpenCL is a framework for writing code for heterogeneous platforms consisting of CPUs, GPUs, DSPs, FPGAs and other processors or hardware accelerators.

4 SELECTED TEST CASES

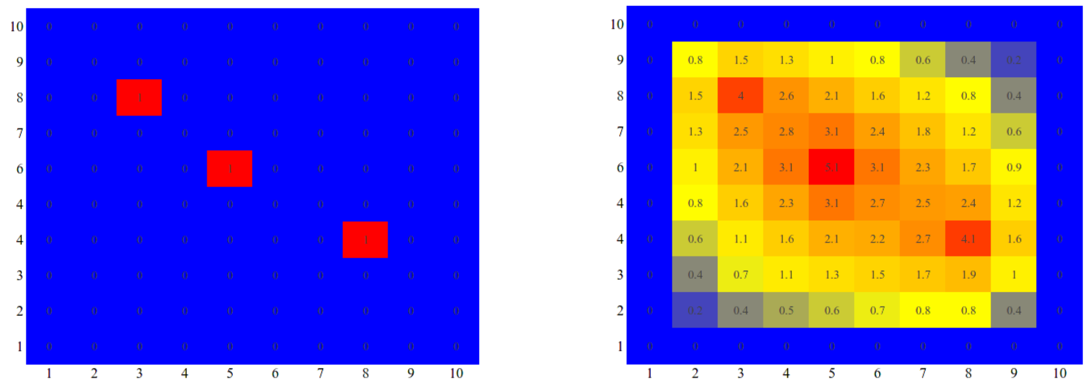
For the present work, some cases considered relevant to the proposed study were selected and addressed. There is a wide range of libraries and/or environments that provide HPC capabilities for Python, as well as there are many possible applications to test them. Given the time limitation, and also the access limitations to clusters and/or supercomputers, this work addresses a limited number of case studies and performs performance analysis for a limited number of approaches to its execution. The following is a brief description of the remaining sections of this chapter, showing its organization:

- section 4.1: The five-point stencil test case (STTC), used in a heat transfer problem on a finite surface modeled by the Poison partial differential equation.
- section 4.2: The fast Fourier transform test case (FTTC), an algorithm that computes the discrete Fourier transform of a sequence.
- section 4.3: The random forest test case (RFTC), an ensemble learning method for classification, regression and other tasks.

4.1 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 4.1), 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 five-point stencil over the 2D domain grid (CHEN et al., 2002) in order to update the temperatures at every time step. An 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 an 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 five-point stencil consists

Figure 4.1 - Initial and final temperature distribution over a finite surface.



(a) Initial zero-temperature distribution over a finite surface exemplified for a 10 × 10 blue-cell grid and the three heat sources arbitrarily chosen, shown as red cells. Two extra columns and rows are included to model the constant zero-temperature border, thus the simulation encompasses only the 8 × 8 inner grid.

(b) Final temperature distribution over a finite surface after 500 iterations, exemplified for same the 10 × 10 grid and three heat sources arbitrarily chosen, shown as red cells, but the simulation encompasses only the 8 × 8 inner grid.

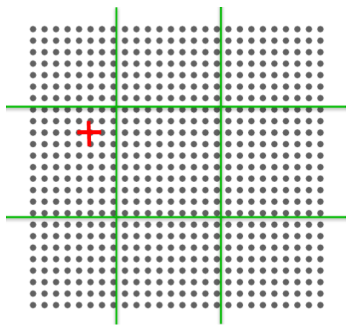
Source: Author's production.

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 five-point stencil is expressed by Equation 4.1.

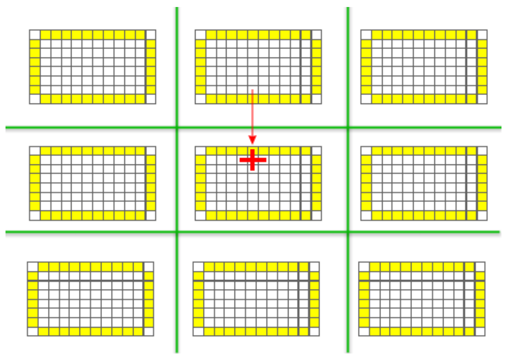
$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} \approx \frac{U_{i+1,j} + U_{i,j+1} - 4U_{i,j} + U_{i-1,j} + U_{i,j-1}}{h^2} \tag{4.1}$$

In the case of parallelization, the domain corresponding to the finite surface is divided into sub domains that are assigned to processes or threads. However, the update of points in the border of sub domains require the temperatures values of neighboring domains, and such data dependency between sub domains implies in message exchange between processes or synchronization between threads (LANGTANGEN; CAI, 2008). Considering the chosen test case, an early implementation of the algorithm was

Figure 4.2 - Discretized 2D domain of the heat transfer case (red cross denotes the five-point stencil).



(a) Discretized 2D domain of the heat transfer problem chosen as test case showing 9 sub domains with their grid points.



(b) Discretized 2D domain of the heat transfer problem divided into 9 sub domains enlarged with their ghost zones.

Source: Adapted from Balaji et al. (2017).

proposed by Balaji et al. (2017), in the C language, but was easily ported to F90. The Figure 4.1(b) 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.

The 2D discrete domain is shown in Figure 4.2(a), with every small circle denotes a grid point and the red cross, the five-point stencil. Green lines show the division of the domain into 9 equal sub domains, for the sake of example. The same domain is shown in Figure 4.2(b), but with each sub domain enlarged by two rows and two columns of extra grid points, shown in yellow, which are copies of the grid points of the four neighboring sub domain. In the borders of the domain, grid points corresponding to the boundary conditions are copied. These extra rows and columns of grid points compose the ghost zone of each sub domain for the specific five-point stencil, with other stencils eventually requiring a different number of rows and columns of grid points. The red arrow denotes the communication/synchronization required to update a grid point of the central sub domain from a “white point” of the sub domain above it, with the temperature of this point copied to the corresponding “yellow point” of the ghost zone of the considered central sub domain.

In the case of a square grid with $N \times N$ points and p processes or threads, each sub domain contains $N/p \times N/p$ points plus its ghost zone composed of two rows and two columns of grid points for the five-point stencil, with a total of $[(N/p)+2] \times [(N/p)+2]$ points per sub domain. The division of a square domain into square sub domains requires then that the number of sub domains be a perfect square. Denoting by p the number of processes or threads, it is convenient that N be divisible by p exactly, considering both as integers.

4.1.1 F90 and Python implementations

Several serial and parallel implementations of the five-point stencil test case described in the previous section 4.1, have been written and executed. Their description and processing performance results are shown ahead. Implementations include F90, 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 (GROPP; LUSK, 1996; University of Tennessee et al., 2020; DONGARRA et al., 1995), which is wrapped by the MPI for Python (`mpi4py`) API (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.

The open-source web application Jupyter Notebook allows the sharing of codes, data and documents that were employed to manage these implementations. It allows to interactively implement and execute the functions that are relative to the specific codes. Serial and parallel of the test case implementations were based on a former work (SOUZA et al., 2018a). Parallelization was performed in this work dividing the square test case problem domain into square sub domains up to the size of 9×9 that is executed by 81 processes.

The Python environment is very flexible and thus it is possible to implement different parallelization schemes. In the case of parallel Cython and Numba, the best scheme in terms of easiness of programming and performance is to encapsulate the most time-consuming part of the specific code in a Python function that will undergo compilation and after, MPI parallelization by means of the `mpi4py` API. The exception here is F2PY API that reuses a binary code generated by a F90 compiler by encapsulating it in a Python function. This work presents some of the most common parallelization approaches for the Python environment, which are described below.

4.1.2 F90 serial and parallel

Obviously, F90 is not a Python implementation for HPC, but is described since it was taken as reference for serial and parallel Python implementations, which are easier and faster to be coded. The F90 serial and parallel versions were compiled with the GNU-compiler gfortran, that complies to the Fortran 95/2003/2008 standard, despite the use of the F90 name here. For instance, Python implementations include F2PY that employs F90 code, and Cython that uses C language code. The serial F90 version is the implementation of the algorithm described in the previous section. Its corresponding parallel version employs the standard MPI asynchronous non-blocking communication functions `MPI_ISend()` and `MPI_IRecv()`, which allows calculations to proceed, while message passing communication is occurring, enhancing the parallel performance. However, at the end of each time step, synchronization is required in a way that each sub domain updates the ghost zones of all neighboring sub domains, in order to have all grid points of all sub domains already updated for the next time step. As already mentioned, for a square grid with $N \times N$ points and p MPI processes, to each process is assigned a sub domain with a total of $[(N/p) + 2] \times [(N/p) + 2]$ points. The time-consuming part of the code updates the domain grid using the five-point stencil as shown in the F90 code in Listing 4.1

Listing 4.1 - Time consuming part of the stencil test case F90 code

```fortran
do j = 2, by+1
 do i = 2, bx+1
  anew(i,j) = (aold(i,j)+(aold(i-1,j)+aold(i+1,j)+aold(i,j-1)+aold(i,j+1))/4.0)
    /2.0
 enddo
enddo
```

4.1.3 Standard Python serial and parallel

The portability of the F90 code to Python is straightforward, without any external library except the NumPy (WALT et al., 2011), which is the Python numerical-tool library. Part of the Python loops can be executed transparently by resources from the NumPy library. The structure and sequence of the original code is preserved and executed interactively by the Python interpreter, and the development and analysis of results is done using the Jupyter Notebook environment. In this way, it is easy to modify the code or its parameters, show and record the results, but given the interpreted-language execution, processing times are high. However, it is easy to perform prototyping in order to check the correctness of the algorithm and to document its code and description. Once such proof of concept of the code is concluded, a further step would be to optimize the code, with a focus in its most computer-intensive parts, which are the performance bottlenecks. 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 by replacing it by a library function. Parallelization is possible employing the Python multiprocessing environment that provides many different ways to provide parallel execution, according to the chosen library. However, in this work, Python multiprocessing was provided by the MPI for Python (`mpi4py`) library, as mentioned in the beginning of this section.

4.1.4 F2PY serial and parallel

F2PY stands for "Fortran to Python" and is a wrapper that creates a Python module by compiling a given F90 source code (PETERSON, 2009). It requires to define a Python function with a list of arguments to be passed for the module execution. In the considered test case, the arguments are the number of grid points, the location and heat rate of the sources, the number of iterations, etc.. In the case of a F90 code already parallelized with MPI, the module can be executed in parallel. However, if the F90 code is not parallelized yet, the typical alternative is to use `mpi4py` (MPI for Python), but it will eventually require to break the original F90 code up into parts and choose the computing-intensive ones to parallelize with MPI. Therefore, F2PY seems more convenient when different Python modules need to be generated and called in an interactive way. The `mpi4py` provides an interface that resembles the Message Passing Interface (MPI) and allows Python to execute a job using one or more shared-memory computer nodes, each one with multi-core processors.

4.1.5 Cython serial and parallel

Cython is a static optimizing compiler for both Python and Cython programming languages (BEHNEL et al., 2010). It is typically employed to create standard Python modules, and also wrapping C/C++ codes. Cython source code is compiled to the C language, which is then compiled again to generate an executable machine code. Each Cython-generated module requires the definition of a corresponding standard Python function. Its processing performance depends on the employed syntactic extensions, standard Python resources and mainly of the choice of specific libraries for some functions. For instance, there are the specific standard C , Python or optimized libraries. Main Python program can then load such Cython-generated functions and execute them. In the case of the test problem serial Python code, all code was compiled by Cython, while in the case of the parallel code, a separate module was created compiling with Cython the double loop that updates the 2D domain, which corresponds to the time-consuming part of the code. There is the corresponding Python function for this module that is then parallelized using `mpi4py`. The time-consuming function that updates the domain grid using the five-point stencil is shown in Listing 4.2.

Listing 4.2 - Time consuming part of the stencil 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] = (aold[i,j]+(aold[i-1,j]+aold[i+1,j]+aold[i,j-1]+aold[i,j
    +1])/4.0)/2.0
```

In the implementation using the Cython compiler, Cython extensions were added to the Python code, as a declaration of “C types” for the variables, and the *flag annotate* is used to generate a compilation output listing, so that it is It is possible to analyze and verify performance improvement points, such as interaction with Python objects and with the Python C API. After manual optimizations, the resulting Cython code allows the compiler to produce most of the code close to the standard C language and its library, and with few dependent parts on the Python library. When using Cython, the final compilation step is transparent and uses the operating system’s standard C compiler to generate the executable code. The development is done in Jupyter Notebook where the compilation is transparent and interactive, and to run the final code in an execution node, it is inserted in a Cython function to be compiled and transformed into a standard Python module, which will be loaded and executed by the Python interpreter on the execution node. Another possible form of development is to use the original Python code without changes, compile using Cython, and in other stages, optimize the code to obtain performance, especially in the parts that are computationally intensive. The increase in serial performance can also be obtained in other ways, and as examples we have: changing the code in Cython in such a way that the compiler does not make connections to the Python interpreter library; use of existing optimized libraries, written in other languages; write critical parts directly in the C or Fortran languages, and use Cython to do *wrapping* in these libraries or modules; rewrite the code using the Cython language, including Cython techniques, resources, and extensions for the purpose of increasing performance.

4.1.6 Numba serial and parallel

The previous approaches are related to using AOT (ahead-of-time) compilers, but Numba is also a JIT (just-in-time) compiler that compiles code at runtime. Numba is compatible with part of the Python language and part of the NumPy module, and uses resources from the LLVM Compiler Infrastructure project, which is a collection of modular and reusable compilers and toolchain technologies that began development in 2000 at the University of Illinois at Urbana-Champaign, and which allows generation of machine code optimized for CPU or GPU (LAM et al., 2015).

In order to compile the test case code, an approach similar to Cython was adopted by creating a Python function that embeds the time-consuming part of the code, and is compiled by Numba. The rest of the Python code is interpreted by standard Python, since it runs fast. In the case of parallel code to be executed in multiple cores/processors/nodes, the Numba-compiled function is also parallelized with `mpi4py`, but execution using a GPU is also possible, since Numba supports part of the Nvidia CUDA API, requiring the definition of the kernel function that will be executed in the GPU. Processing performance for the Numba-GPU version is shown ahead in subsubsection 5.2.2.1. In the case of the standard CPU-bound Numba code, the time-consuming function that updates the domain grid using the five-point stencil is shown in Listing 4.3.

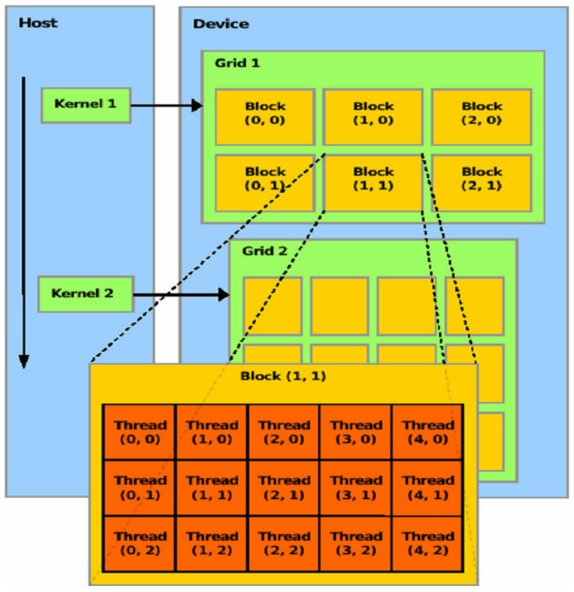
Listing 4.3 - Time consuming part of the stencil test case Numba code

```python
@jit(nopython=True)
def kernel(anew, aold) :
  anew[1:−1,1:−1] = (aold[1:−1,1:−1]+(aold[2:,1:−1]+aold[:−2,1:−1]+aold[1:−1,2:]+aold[1:−1,:−2])/4.0)/2.0
```

4.1.7 Numba-GPU

Execution using a GPU is also possible, since Numba supports part of the Nvidia CUDA API, requiring the definition of the kernel function that will be executed on the GPU. Numba-GPU uses a wrapper for the Cuda API in C language, providing access to most of its features. The Numba-GPU implementation required more modifications compared to standard Python serial code, than other implementations. Likewise, the calculation of the stencil and the consequent updating of the grid points, makes up the main loop of the test case algorithm. This loop was encapsulated in a function with a Numba decorator to run on a single GPU using compilation

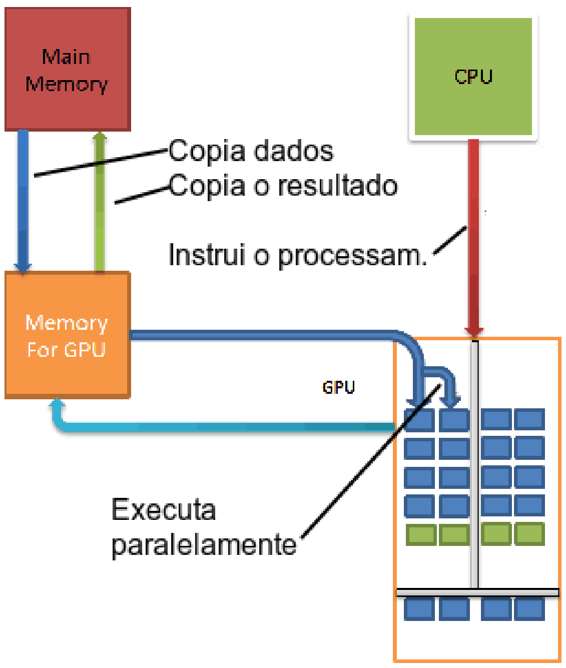
Figure 4.3 - Execution illustrating blocks of 15 threads



Source: Adapted from Daniel and Mircea (2010).

just-in-time (JIT). The single and multiple threads (SIMT) instruction paradigm models the execution of the GPU, with the problem domain mapped in a block grid composed of a number of threads. The blocks are then divided into warps, usually 32 threads, and executed by one of the streaming multiprocessors that make up the GPU (Figure 4.3). As shown in Figure 4.4, the arrays are copied from the main memory to the GPU memory, and at the end the result is copied from the GPU memory to the main memory. The remaining code of the test case algorithm is executed by the standard Python interpreter, since it does not require computationally intensive use.

Figure 4.4 - GPU processing flow



Source: Adapted from Li et al. (2015).

4.2 The fast Fourier transform test case

This section describes the implementation of the Fast Fourier Transform (FFT) algorithm, which is an numerical algorithm for converting a finite sequence of equally spaced samples, into a sequence of the same size, also equally spaced, using a discrete time Fourier transform (DTFT, a form of Fourier analysis that is applicable to a sequence of values), and the result is a function of the frequency that uses complex values, representing the frequency domain of the original input sequence. In mathematics the discrete Fourier transform is called DFT and sometimes the terms FFT and DFT, although they are not the same thing, are used interchangeably. The transformation discretizes and transforms, and is used in sums of simple trigonometric functions to represent or approximate general functions in the analysis of practical applications. In other words, the algorithm converts a sampled function from the original time domain to the frequency domain. It can be used, for example, to process and analyze a radio signal sample over a period of time, moving from the time domain to the frequency domain. A signal is generally a discretized input for the transformation, and the spectrum or transformation in the frequency domain is usually the output. The definition used in this implementation is given by the Equation 4.2.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \quad , \qquad (k = 0, ..., N-1) \tag{4.2}$$

Where:

N:	size of the sequence
x_n:	input sequence of N complex numbers $\{x_n\} := x_0, x_1, ..., x_{N-1}$
X_k:	output transformed sequence of complex numbers $\{X_n\} := X_0, X_1, ..., X_{N-1}$
i:	$\sqrt{-1}$, the imaginary part notation

Complex inputs and outputs are used to define the transform, and if necessary, a single linear frequency f component can be represented by a complex exponential $x_n = e^{i2\pi f n \Delta t}$, where Δt is the interval of sampling. The inverse DFT is not used in this implementation.

To illustrate, the Python serial implementation uses the NumPy library's FFT module, and the convention used is `X = fft (x, N)` with symbols similar to those used in the equation. In this module, `X[0]` returns the sum of the signal, `X[1:N/2]` (the first half of the output sequence) returns the positive-frequency terms, and `X[N/2+1:]`

(the second half) returns the negative-frequency terms, in order of decreasing negative frequency. For an even number of input points, `X[N/2]` represents both positive and negative Nyquist frequency (or folding frequency). For an odd number of input points, `X[(N-1)/2]` contains the largest positive frequency, while `X[(N+1)/2]` contains the largest negative frequency.

The calculation of multi-dimensional FFT is done as the composition of a sequence of one-dimensional FFTs, along each dimension. For example, a multidimensional 3-D FFT for a sequence of complex numbers, with a total of N^3 elements, can be computed by dividing by columns or rows, and using one-dimensional 1-D FFT (N^2 elements) to carry out the transformation. The output sequence will be the junction of the independent transformations. Note that, unlike the stencil case, there is no dependence and communication between neighboring cells necessary for the calculation of a cell. For simplicity, this implementation considers the size of the dimensions equal, in all sequences of numbers. So if a given sequence size is N, the dimensions L, M, and N are equal to N.

For the test case, the multidimensional 3D FFT was chosen, and to generate the input sequence, although there are countless ways as well as different applications to use, for simplicity a simple quadratic equation was chosen, which takes into account the coordinates in the three axes of the input matrix (Equation 4.3), so that the result is always the same for a given sequence size, and in this way it is possible to verify in a simple way the correct functioning of the code for different implementations of the algorithm. In the case of multiprocessing, the equation also simplifies division, distribution, and communication, since there is no large multidimensional array to share between processors, the most time spent on processing is transformation time.

$$x_{l,m,n} = (lN^2 + mN + n + 1) \cdot 10^{-6} \tag{4.3}$$

Where $l = (0, 1, ..., N-1)$, $m = (0, 1, ..., N-1)$, and $n = (0, 1, ..., N-1)$

After calculating the FFT and obtaining the output sequence, a checksum is generated to facilitate the identification of any problems in the implementation of the algorithm, and to allow easy comparison with other results that have a sequence of the same size (Equation 4.4). In the case of multiprocessing, there is a short communication time after transformation, to send and receive the checksum calculated in each process, which is then used to obtain the total sum.

$$S = \sum_{l=0}^{N-1} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x_{l,m,n} \tag{4.4}$$

Parallel FFTs are computed through a combination of division into parts, distribution and sharing of multidimensional matrices or arrays between a series of different processors, making the transformation of each part and, finally, joining each resulting part. The divided parts can be slabs that correspond, for example, to the sliced dimensions of a matrix. In one example, if a 3-D matrix is composed of three 2-D matrices and we have three processors, you can split the matrix or array into slabs and share it between each processor, which receives a slab composed of a 2-D matrix, does the transformation, and the result of each slab is attached to the output sequence. There are other forms of composition, distribution, and computation, but for simplicity, and to follow the existing implementation and documentation, an attempt was made to follow a standard implementation of the library used in the code.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.2.1 F90 and Python implementations

The organization and some aspects of this subsection are similar to subsection 4.1.1 of the previous case study. Several serial and parallel implementations of the Fast Fourier Transform test case described in section 4.2 were written and executed and their descriptions and processing performance results are shown below. The implementations include F90, taken as a reference although in some cases F2PY has a slightly superior performance, Python, F2PY, Cython, Numba, Numba-GPU, and using the MPI library (encapsulated by `mpi4py` or the FFT library used) that allows the use of one or more computer nodes.

The open source web application Jupyter Notebook was used to experiment, develop, execute and analyze the results, including the F90 implementation. It allows the sharing of codes, data and documents that were used to manage these implementations and allows interactive code-related functions to be implemented and executed interactively, as well as aiding in the reproducibility of obtaining the results. The parallelization was performed by dividing the 3-D array that represents the input sequence, with dimension size 576x576x576, and the calculation is then performed by up to 81 processes in up to 4 execution nodes with up to 24 processes per node, in the case of the Bull B710 node. The size of the sequence was chosen in such a way that the execution time of the implementation was around 20 s, so that it was close to the times of the test case of the five point stencil.

The Python environment is very flexible and allows parallelization in several ways. In the case of standard Python, Cython, and Numba, the `mpi4py_fft` library was used, which is a package for FFT computing that uses the FFTW library (section 3.12), and which includes the `mpi4py` library in itself, allowing the `mpi4py_fft` library to automatically distribute and communicate large sequences or arrays. In the case of the F90 and F2PY implementations, the FFTW library and the MPI library were used. F2PY does not use the `mpi4py` library, and reuses the same F90 code with FFTW and MPI. This work presents some of the most common parallelization approaches for the Python environment, which are described below.

4.2.2 F90 serial and parallel

The organization and some aspects of this subsection are similar to the subsection 4.1.2 section of the previous case study. The F90 is described as it was taken as a reference for serial and parallel Python implementations. The serial and parallel F90 implementations were compiled with GNU Fortran, which is compliant with the Fortran 95 standard and has features from Fortran 2003 and 2008, despite the use of the name F90 here. The F90 serial implementation is the implementation of the algorithm described in section 4.2. Its corresponding parallel version employs the communication functions of the MPI library, which is declared in the code body and used internally by the FFTW library. A 3-D array of complex numbers, of equal dimensions M = N = L is used to store the input sequence. A quadratic function that considers the dimensions' indices $(lN^2 + mN + n + 1)$ is used to generate the sequence and store it in the array, and thus in this implementation there is no need to communicate slices of input sequences between processes. Although not used in this case, when necessary, the FFTW library distributes data in blocks of one dimension

(1d) distributed along the first dimension. For example, for a transformation using an array of complex numbers with dimensions L x M x N, distributed over P MPI processes, each process will have $(L \div P)$ x M x N data slices. In the implementation made, although there is no communication of slices, the division into slices is used in the quadratic equation to calculate the input sequence. The FFTW library also has a planning step (plan) before transformation to determine what is the best way to carry out the transformation. The time-consuming part of the code is shown in Listing 4.4, and is mostly composed of the transformation computation (fftw_mpi_execute_dft) and to a lesser extent by the generation of the input sequence (data = myfunction ...).

Listing 4.4 - Time consuming part of the FFT test case F90 code

```
! initialize the 3-D data to some function.
data = myfunction ...
! compute transform in 'data'. 'plan' is the way to calculate.
call fftw_mpi_execute_dft(plan, data, data)
```

4.2.3 Standard Python serial and parallel

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.2.4 F2PY serial and parallel

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus

vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.2.5 Cython serial and parallel

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.2.6 Numba serial and parallel

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.2.7 Numba-GPU

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu

libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.3 The random forest test case

For the random forest test case, Python was used with the Scikit-learn library, which does most of the heavy lifting. As the library is compiled using Cython or C/C++, and optimized for performance and parallelism, as well as being actively developed, the results are better than using F90 with a library that is not as optimized and not actively developed. Note that in this case, although we use the expression "Python" to reference the implementation, we are actually using Python's interfacing feature to run a program in machine code, compiled and optimized. This way the performance is greater than if the standard Python interpreter were used to execute the code that requires computational effort.

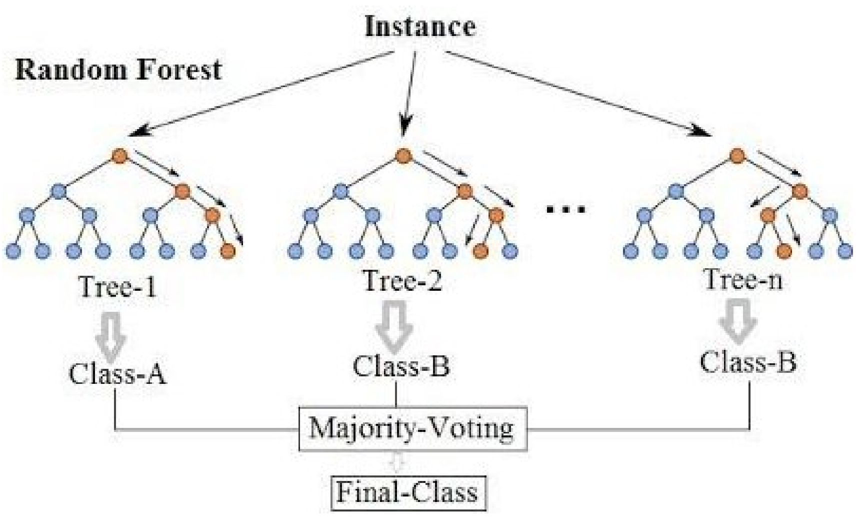
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

The F90 and F2PY implementations use the PARF library (**??**) which is not as optimized as Scikit-learn, nor actively developed, and the results showed lower performance for the developed implementation.

As most of the work is done by the used libraries, it was not necessary great programming efforts, which were concentrated in configuring parameters and dataset, and in the parallel case, establishing the parallelization environment.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Figure 4.5 - Randon Forest



Source: DIMITRIADIS, S. et al. How random is the random forest?

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.3.1 F90 and Python implementations

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu

libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.3.2 F90 serial and parallel

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.3.3 Standard Python serial and parallel

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.3.4 F2PY serial and parallel

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.3.5 Cython serial and parallel

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.3.6 Numba serial and parallel

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.3.7 Numba-GPU

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.4 Profiling

Profiling was performed on the F90 and F2PY stencil and FFT implementations. For F90, the Intel Application Performance Snapshot (APS) was used, and for F2PY the profiling features of IPython and Jupyter Notebook were used. As F2PY reuses the F90 code and turns it into a Python library, the analyzes are complementary. In the executed implementations, the library performs the greatest computational effort, including message exchange using MPI. The profiling analysis is more in-depth in F90, and in the case of F2PY the upper layer (Python) is analyzed.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.4.1 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.4.2 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.4.3 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.4.4 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.4.5 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.4.6 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.4.7 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat.

Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5 ANALYSIS OF THE PARALLEL PERFORMANCE OF TEST CASES

This chapter covers the analysis of the three executed case studies, and performs performance analyzes for a limited number of approaches to their executions, showing performance in the form of runtimes, speedups, and parallel efficiencies. There are implementations for CPU and GPU, in sequential and parallel versions using MPI. For this work no implementations were made using threads (OpenMP). Case studies were analyzed separately, no comparisons were made between case studies. The following is a brief description of the other sections in this chapter, showing their organization, which follows the same sequence as the previous chapter:

- section 5.1: XXXX
- section 5.2: The five-point stencil case, a heat transfer problem over a finite surface. XXXX
- section 5.3: The fast Fourier transform case, an algorithm that computes the discrete Fourier transform of a sequence. XXXX
- section 5.4: The random forest case, an ensemble learning method for classification, regression and other tasks. XXXX

5.1 Analysis introduction

5.1.1 Speedup and efficiency

Standard HPC performance metrics are used here like the speedup $\mathtt{S}_p$, given by the ratio between the serial execution time $\mathtt{t}_s$ and the parallel execution time $\mathtt{t}_p$, using p processes or threads (Equation 5.1), being the ideal speedup, called linear speedup, equal to p.

$$S_p = \frac{t_s}{t_p} \tag{5.1}$$

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 $\mathtt{E}_p$, given by the ration between the speedup and the corresponding number p of processes or threads (Equation 5.2). Thus, the linear speedup is equivalent to the unit or 100% efficiency.

$$E_p = \frac{sec_p}{p} \tag{5.2}$$

5.1.2 Computer-system environment

In this chapter we will cover the computational tests performed for the chapter 4 test cases, related to the algorithms:

a) The five-point stencil case;

b) The fast Fourier transform case;

c) The random forest case.

For the execution of the models, the parameters used in all tests were:

a) The processing performance of the different implementations of the heat transfer test uses a square grid of 4800 x 4800 points and 500 time steps were adopted, with the three constant-rate heat sources;

b) The processing performance of the different implementations of the fast Fourier transform test uses a three-dimensional grid with dimensions 576 x 576 x 576, and each point on the grid is a complex number generated by a mathematical function that uses the point coordinates in the calculations;

c) The random forest case uses 66000 instances for the training set, and 34000 for the test set. Other sizes were also used in particular cases discussed later.

5.1.3 Computer nodes

Different computer nodes of the LNCC (National Laboratory for Scientific Computing) SDumont (Santos Dumont) supercomputer were employed:

- **Bull thin nodes B710**, each one with 2 Intel Xeon E5-2695v2 Ivy Bridge (2.4 GHz) 12-core processors (total of 24 cores per node) and 64 GB main memory; compilers and libraries include GNU Fortran 7.4, GNU Fortran 8.3, OpenMPI 4.0.1, Intel Fortran 19.0.3, Intel MPI, Python 3.6.12, Cython 0.29.20, NumPy 1.18.1, and Numba 0.41.0;

- **Bull thin node B715**, with 2 Intel Xeon E5-2695v2 Ivy Bridge (2.4 GHz) 12-core processors (total of 24 cores per node), 64 GB main memory, and 2 GPUs Nvidia Tesla K40t; in addition to the compilers and libraries of the Bull B710 nodes, there is CUDA 10.1;
- **Bull Sequana X node**, with 2 Intel Xeon Gold 6152 (2.1 GHz) 22-core processors (total of 44 cores per node), 4 GPUs Nvidia Volta V100 and 768 GB main memory; in addition to the compilers and libraries of the Bull B710 nodes, there is CUDA 10.1.

5.1.4 F90, versions, flags, and run nodes

Table 5.1 - Processing times as a function of the number of processes in the F90 implementation on a single computer node (Bull B710 or Sequana-X) for the versions compiled with GNU/gfortran (OpenMPI) and Intel/ifort (Intel MPI).

	Seq	**1**	**4**	**9**	**16**	**36**
Bull Seq-X/GNU	15.82	**15.64**	**4.13**	**2.09**	**1.48**	**1.21**
Bull Seq-X/Intel	**14.81**	15.67	5.03	2.94	2.47	2.45
Bull B710/GNU	19.25	21.91	7.34	6.15	4.68	-
Bull B710/Intel	21.87	20.66	7.32	6.21	4.63	-

Source: Author's production.

In all F90 and Python serial and parallel implementations of this work, the GNU 4.8.5 suite of compilers and the OpenMPI 4.0.1 (MPI library) were employed. The suite includes the gfortran or gcc compilers, used according to the case, both with the -O3 optimization flag, which is sometimes cited in the literature as a good balance between run-time performance and compilation time in most cases. Flags are used by compilers to mark something to attract attention or to modify the default behavior of the compiler, and the -O3 flag instructs the compiler to perform a series of optimizations in the generated code, which in general means more performance to the detriment of the generated code looking more different from the original source code (this may not be desired during the debugging step, for example). It is important to note that some flags can change the default behavior of the compiler in a way that may not be desired in some cases, unwanted effects may occur, the program may not behave as expected in all situations, and therefore it is necessary to use them carefully. Note that F2PY (section 3.8) by default also uses the -funroll-loops flag, which tells the compiler to undo loops whose number of iterations can be determined at compile time or at the loop entry. As -O3 and -funroll-loops are standard for F2PY (it is not necessary to include), for this work we chose to leave it this way for simplicity. Only in the case of F90 the compilation flag -O3 was added.

In the case of F90 serial and parallel implementations, also the version compiled with the Intel ifort 19.1.2 and the Intel MPI 19.1.2 library was tested, with the -O3 optimization flag. However, performance results were similar to the ones obtained with GNU Fortran, as it is shown in Table 5.1, and thus only results of the F90 compiled version with the GNU Fortran with OpenMPI are shown in the rest of this work. It is also important to note that in the case of the Bull Sequana X node, only one was available and, therefore, in general, tests on one and several nodes were performed on Sequana B710 nodes.

5.2 Analysis of the five-point stencil test case

This section shows the processing performance of the five-point stencil test case, for the serial and parallel implementations performed on CPUs and GPUs, and is divided into two subsections, one for CPU, and the other for GPU.

5.2.1 Processing performance of implementations executed by the CPU

This subsection 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. The Table 5.2 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 5.1 also shows processing times in function of the number of MPI processes for the different implementations, and Figure 5.2 shows the corresponding speedups calculated taking the F90 serial time as reference.

In general, according to Table 5.2, 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 5.2, Figure 5.1 and Figure 5.2, 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

Table 5.2 - Performance of the different implementations of the stencil 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								
	Serial	**1**	**4**	**9**	**16**	**36**	**49**	**64**	**81**
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								
	Serial	**1**	**4**	**9**	**16**	**36**	**49**	**64**	**81**
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

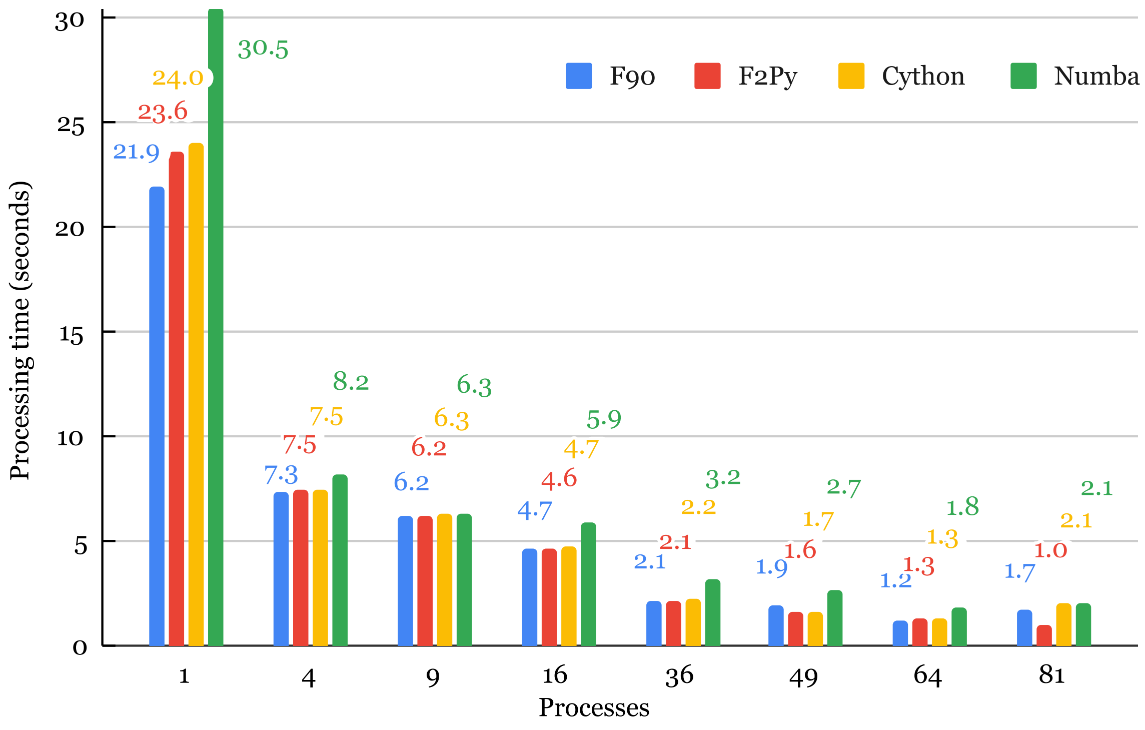
Source: Author's production.

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, which require 3 computer nodes. The exception was F2PY that obtained the lowest time with 81 MPI processes.

5.2.1.1 F90 serial and parallel performance

In this work, the parallelization was performed by dividing the domain of the test case into square sub-domains of up to 9×9 that is performed by 81 processes. The F90 implementation, serial, was used as a reference for evaluating the other serial

Figure 5.1 - Processing times (s) of the different implementations of the stencil test case, depending on the number of processes (for convenience, times above 30 s were not fully depicted).

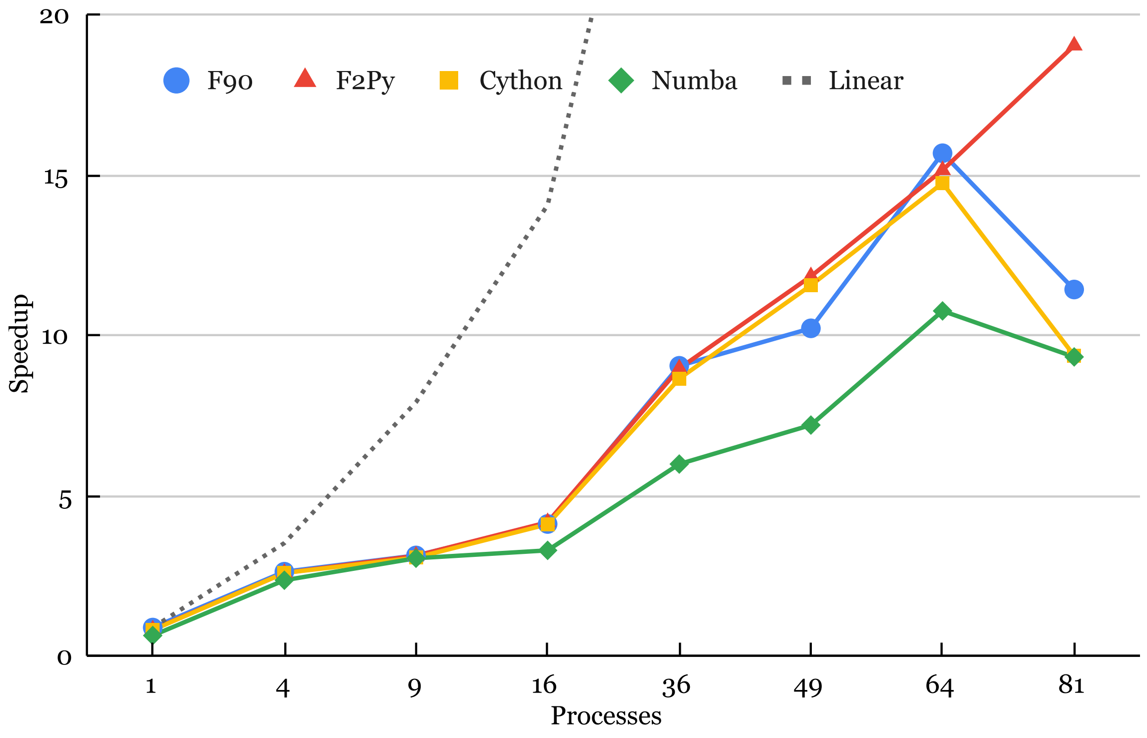


Source: Author's production.

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. F90 obtained the best serial 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.

The results are evaluated in the Jupyter Notebook interactive environment, using the Cell magics *%%writefile* to write the source code without changes, and the Cell magics *%%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. As a suggestion for a

Figure 5.2 - Speedups of the different implementations of the stencil 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. Dashed line denotes the linear speedup.



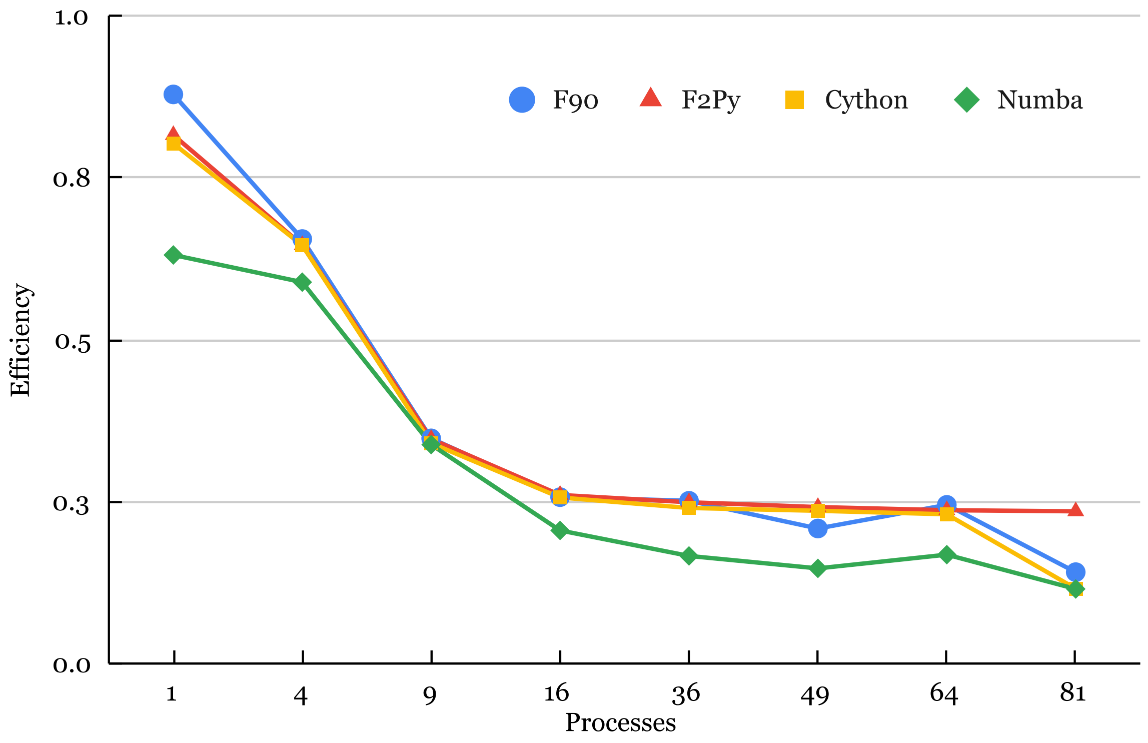
Source: Author's production.

future study, it would be interesting to investigate why F90 did not get the best time under all conditions.

5.2.1.2 Standard Python serial and parallel performance

The serial implementation in Python, in direct conversion, without the use of external libraries 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 those

Figure 5.3 - Parallel efficiencies of the different implementations of the stencil test case, depending on the number of processes.



Source: Author's production.

parts that need 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.

5.2.1.3 F2PY serial and parallel performance

F2PY, together with F90, were the best performing implementations. Despite the overhead added by the wrapper and the Python interpreter, F2PY achieved superior performance for 16, 49 and 81 processes, and also in the serial implementation. For 81 processes, the performance of F2PY was much better than F90 (1.01 s versus 1.69 s).

F2PY reuses the F90 code and creates a standard Python module, which is then used in the main standard Python code. 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 standard Python main code. Jupyter Notebook is used in all stages, including sending for execution using the queue manager and the scheduler, and reading the resulting output files. As a suggestion for future studies, it would be interesting to better understand why the performance differences occurred, especially between F90 and F2PY. In this case, it is important to note that both use the -O3 compiler flag, but F2PY by default also uses the -funroll-loops flag (see subsection 5.1.4).

5.2.1.4 Cython serial and parallel performance

The performance of the serial Cython implementation is between F2PY and Numba, and the parallel is close to F90 and F2PY from 4 to 64 processes. Cython in the serial 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. 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.

5.2.1.5 Numba serial and parallel performance

Numba's performance was below F90, F2PY, and Cython, but well above standard Python. The performance of the serial implementation was worse than serial Cython, and in the parallel implementation the performance was close to or equal to Cython from 4 to 81 processes.

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 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 that it can also be used for processing using GPU, if applicable. And since the build is JIT, the code can eventually be optimized during execution making the implementation portable without the need for previous AOT builds for each architecture, before execution.

5.2.1.6 Overhead

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5.2.2 Processing performance of the Numba-GPU implementations

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5.2.2.1 Single-process and single-core Numba-GPU

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 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.

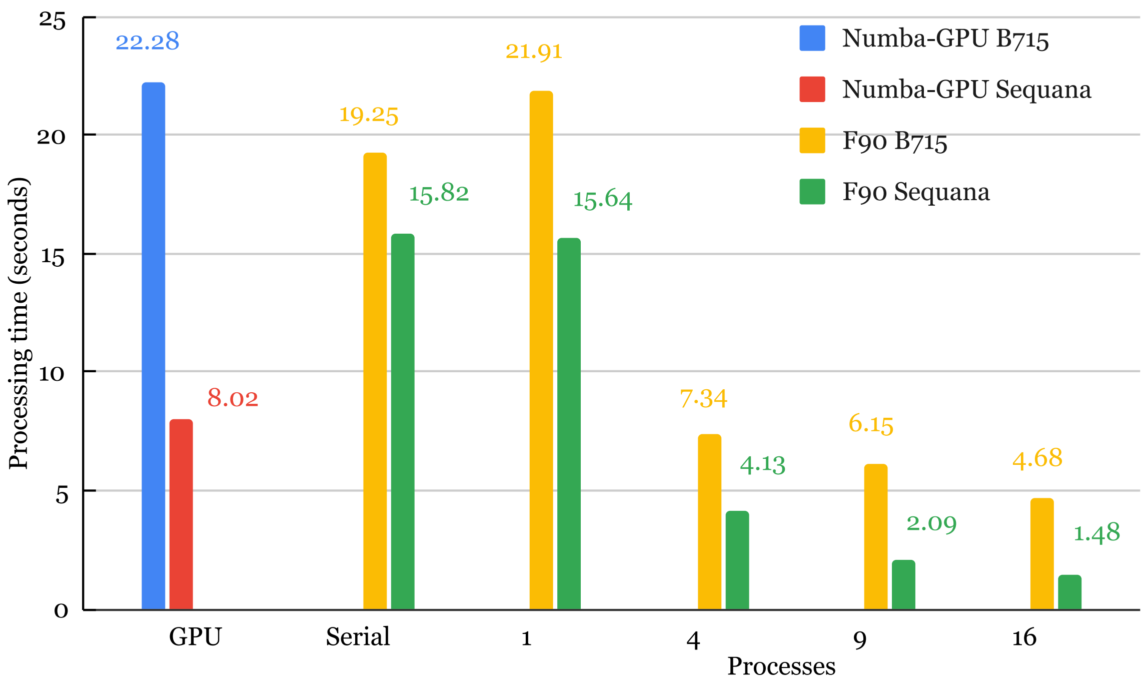
Table 5.3 - Performance as a function of the number of processes for the F90 and Numba-GPU implementations of the stencil test case executed on the Bull Sequana-X and on the Bull B715 node. The execution time of the F90-compiled serial code in the Bull B715 node was taken as the reference for the speedup calculation (highlighted in **blue**).

	Processing time (s)					
	GPU	**Serial**	**1**	**4**	**9**	**16**
F90/B715		**19.25**	21.91	7.34	6.15	4.68
F90/Seq-X		**15.82**	**15.64**	**4.13**	**2.09**	**1.48**
Numba-GPU/B715	22.28					
Numba-GPU/Seq-X	**8.02**					
	Speedup					
	GPU	**Serial**	**1**	**4**	**9**	**16**
F90/B715		**1.00**	0.88	2.62	3.13	4.11
F90/Seq-X		**1.22**	**1.23**	**4.66**	**9.22**	**13.02**
Numba-GPU/B715	0.86					
Numba-GPU/Seq-X	**2.40**					
	Parallel efficiency					
	GPU	**Serial**	**1**	**4**	**9**	**16**
F90/B715		**1.00**	0.88	0.66	0.35	0.26
F90/Seq-X		**1.22**	**1.23**	**1.17**	**1.02**	**0.81**
Numba-GPU/B715	0.86					
Numba-GPU/Seq-X	**2.40**					

Source: Author's production.

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 (Figure 5.4). 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 8×8 threads in the case of the V100 GPU (Bull Sequana-X node) and blocks of 16×16 threads in the case of the K40 GPU (Bull B715 node). These block sizes were optimized by trial-and-error.

Figure 5.4 - Processing times (s) as a function of the number of processes for the F90 and Numba-GPU implementations of the stencil test case executed on the Bull Sequana-X (blue and purple bars) and on the Bull B715 node (red and green bars).

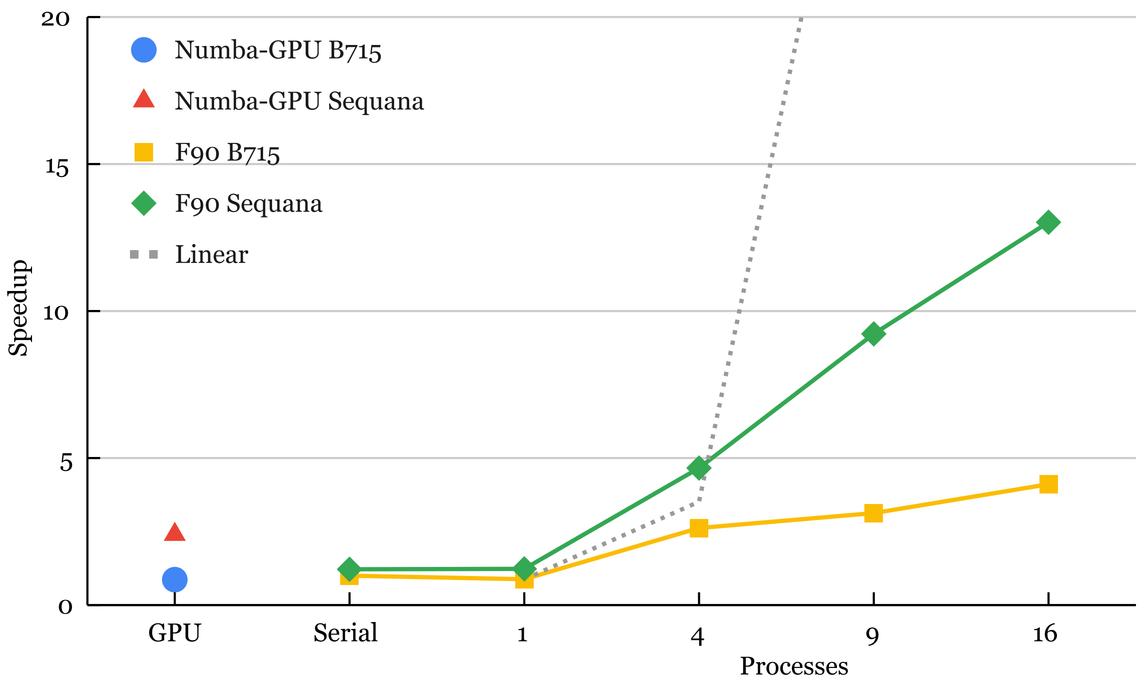


Source: Author's production.

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 shown in Table 5.3 and in Figure 5.4, Figure 5.5, and Figure 5.6. 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

Figure 5.5 - Speedups as a function of the number of processes for the F90 and Numba-GPU implementations of the stencil test case executed on the Bull Sequana-X (blue triangle and purple line) and on the Bull B715 node (red dot and green line). The execution time of the F90-compiled serial code in the Bull B715 node was taken as the reference for the speedup calculation. Dashed line denotes the linear speedup.



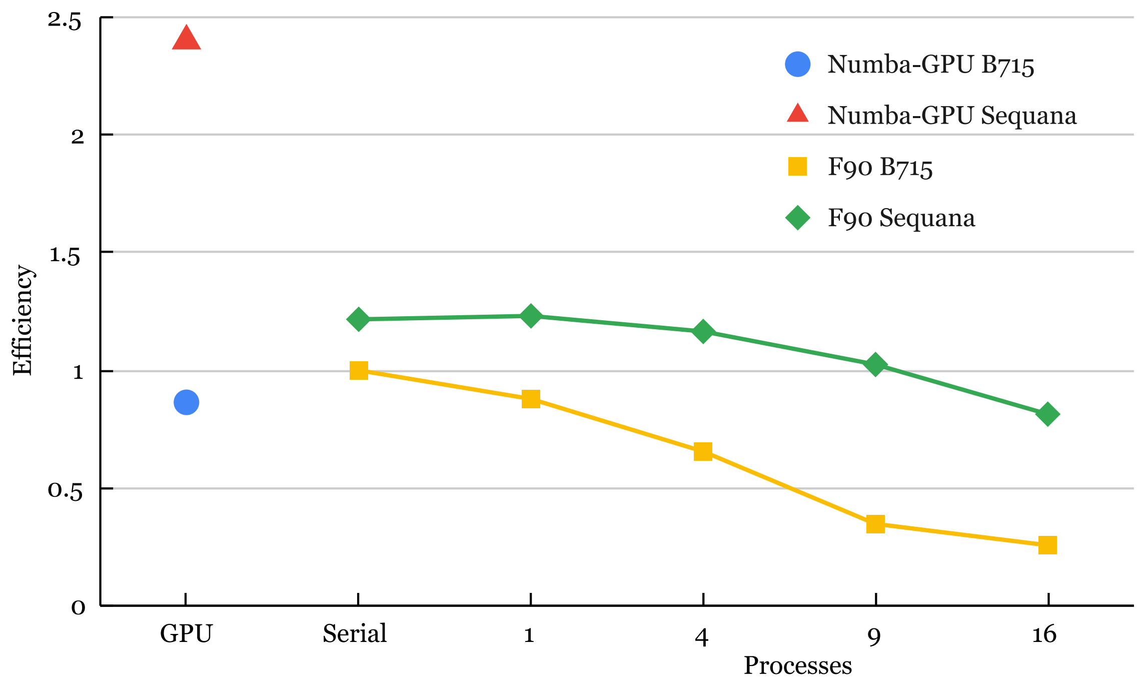
Source: Author's production.

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 multi-core and multi-GPU 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 Bull Sequana-X node shows the processing power of such accelerators, since it is 2.4 times faster than the F90 serial implementation.

5.2.2.2 Multi-process and multi-core Numba-GPU

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu

Figure 5.6 - Parallel efficiencies as a function of the number of processes for the F90 and Numba-GPU implementations of the stencil test case executed on the Bull Sequana-X (blue triangle and purple line) and on the Bull B715 node (red dot and green line).



Source: Author's production.

neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5.3 Analysis of the fast Fourier transform test case

This section shows the processing performance of the fast Fourier transform test case in much the same way as in the previous section 5.2, divided into two subsections, one for CPU and another for GPU. During the analysis of the implementations, the repetition of the same texts that are in common with the case described in the previous section will be avoided, but this will be indicated with the reference to the previous section. If desired, please read the reference provided to complete with more information.

5.3.1 Processing performance of implementations executed by the CPU

This subsection 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. The Table 5.4 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, 16, 24, 48, 72, and 96 processes in the Bull B710 nodes. Figure 5.7 also shows processing times in function of the number of MPI processes for the different implementations, and Figure 5.8 shows the corresponding speedups calculated taking the F90 serial time as reference.

According to Table 5.4, the shortest time obtained was 0.94 s with 16 MPI processes. F2PY serial obtained a slightly better time than F90 serial, however to standardize the analysis and be similar to the previous section, the F90 time was used as a reference for speedup and efficiency calculations. In general, the F2PY implementation achieved the best performance[1], with the shortest processing time of 0.96 s and 16 MPI processes, close to F90. Then, in a general way, comes the F90 implementation, with the shortest time being 0.94 s previously mentioned. Numba, for 9 MPI processes, presented the best time, 2.41 s. F2PY required relatively few changes to the original F90 code, while Cython and Numba required rewriting the F90 code translating to Python, or writing new code using Python. To achieve performance in Cython, more code changes are needed than Numba. Numba's performance, in general, was better than Cython and Python, but worse than F2PY. The very low performance of the standard Python serial and parallel versions shows the convenience of using

[1]Note that F2PY by default also uses the -funroll-loops flag (see subsubsection 5.2.1.3).

Table 5.4 - Performance of the different implementations of the FFT test case described, depending on the number of MPI processes: processing times (seconds), accelerations and parallel efficiencies. The best values for serial or for each number of MPI processes are highlighted in **red**. The execution time of the serial code compiled with GNU-Fortran was taken as a reference for the calculation of speedup (in **blue**).

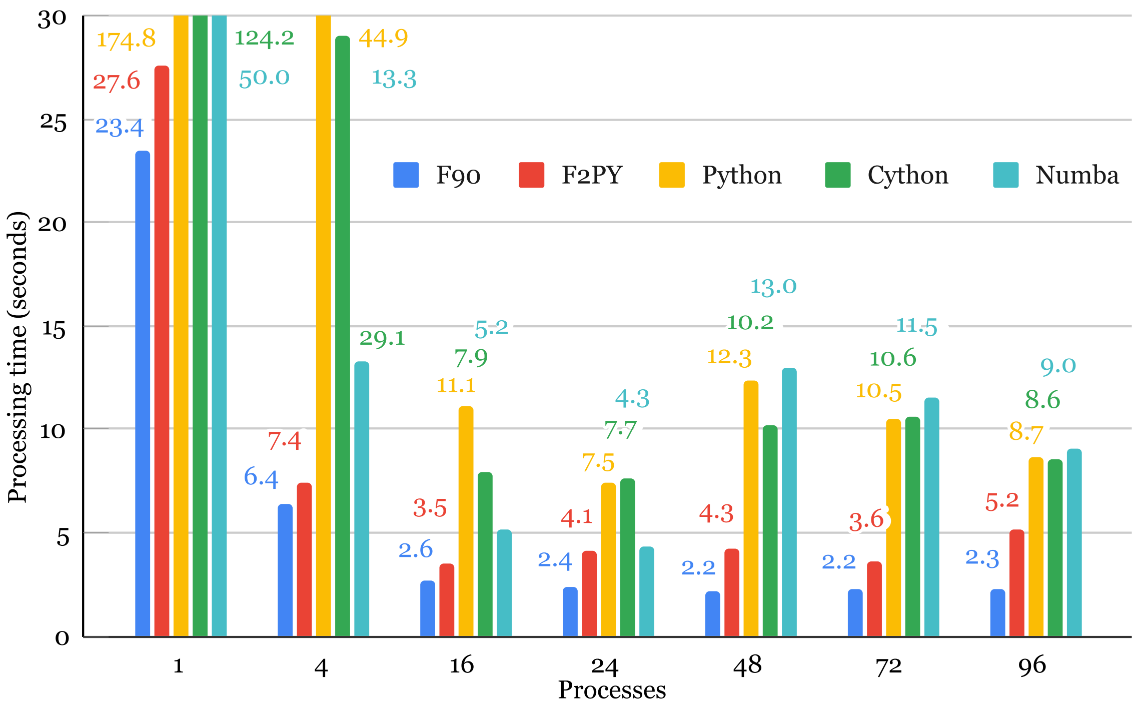
	Processing time (seconds)							
	Seq.	**1**	**4**	**16**	**24**	**48**	**72**	**96**
F90	**19.29**	**23.43**	**6.42**	**2.65**	**2.36**	**2.22**	**2.25**	**2.33**
F2PY	23.83	27.61	7.42	3.53	4.09	4.28	3.60	5.19
Python	161.70	174.78	44.88	11.09	7.47	12.32	10.47	8.68
Cython	109.03	124.17	29.06	7.92	7.66	10.21	10.62	8.57
Numba	48.64	50.01	13.25	5.20	4.33	13.00	11.53	9.04
	Speedup							
F90	**1.00**	**0.82**	**3.00**	**7.28**	**8.18**	**8.70**	**8.58**	**8.28**
F2PY	0.81	0.70	2.60	5.47	4.72	4.51	5.36	3.72
Python	0.12	0.11	0.43	1.74	2.58	1.57	1.84	2.22
Cython	0.18	0.16	0.66	2.43	2.52	1.89	1.82	2.25
Numba	0.40	0.39	1.46	3.71	4.45	1.48	1.67	2.13
	Parallel efficiency							
F90	**1.00**	**0.82**	**0.75**	**0.46**	**0.34**	**0.18**	**0.12**	**0.09**
F2PY	0.81	0.70	0.65	0.34	0.20	0.09	0.07	0.04
Python	0.12	0.11	0.11	0.11	0.11	0.03	0.03	0.02
Cython	0.18	0.16	0.17	0.15	0.10	0.04	0.03	0.02
Numba	0.40	0.39	0.36	0.23	0.19	0.03	0.02	0.02

Source: Author's production.

implementations like F2PY, Cython, or even Numba. As the reference time for calculating the speedup was the same F90 serial time for all implementations, the classifications for speedups and parallel efficiencies are similar to the classifications for processing times.

As can be seen in Table 5.4, Figure 5.7 and Figure 5.8, the parallel scalability is poor, since the test case algorithm decomposes the multidimensional matrix and distributes between processors, requiring an amount of communication between processes that decreases parallel efficiency to less than 40% for 9 MPI processes or more, in the case of most of the implementations shown. It can be seen that for up to 16 MPI processes, executed on Bull B710 nodes, all implementations had performed similarly. In the case of 36 MPI processes, which requires 2 execution nodes, the performance

Figure 5.7 - Processing times, without seconds, for the different implementations of the FFT test case, depending on the number of processes (for convenience, times above 30 s do not appear on the graph)



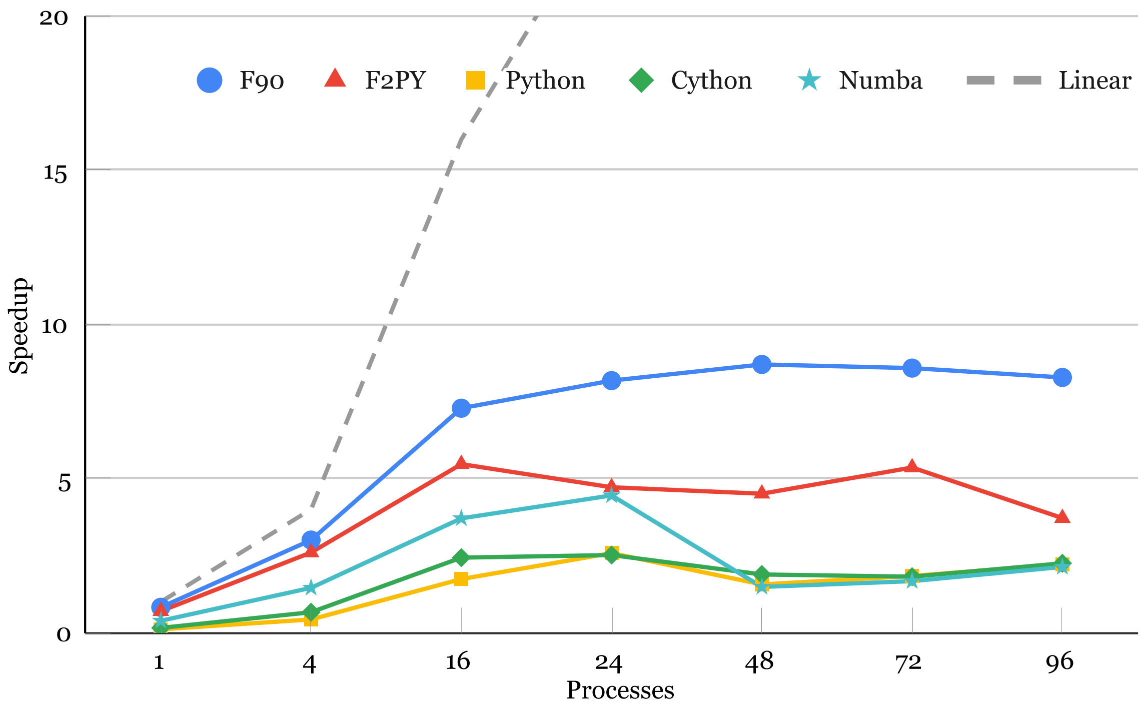
Source: Author's production.

of all implementations suffered a significant drop compared to 16 MPI processes, which requires 1 computing node. The exceptions were F90 and F2PY with 36 MPI processes, which achieved parallel efficiency of approximately 20%, above the others but with a tendency to approach the other implementations especially in terms of efficiency.

5.3.1.1 Comments on obtained F90 serial and parallel performance

In this work, parallelization was carried out dividing the multidimensional matrix and distributing between processes, which is executed by up to 81 processes. The serial F90 implementation was used as a reference to evaluate the other serial implementations, for execution in a core, without parallelization. The implementation does not depend on Python and uses a Fortran compiler. It uses as parameters of execution, the dimension of the multidimensional matrix, which is the same for all dimensions. F90 in general obtained practically the same execution time as F2PY, or close values, and the two implementations obtained the best serial times among the serial

Figure 5.8 - Speedups of the FFT test case implementations as a function of the number of processes. The runtime of the compiled serial GNU-Fortran code was used as a reference for the calculation. Dotted lines denote linear speedup



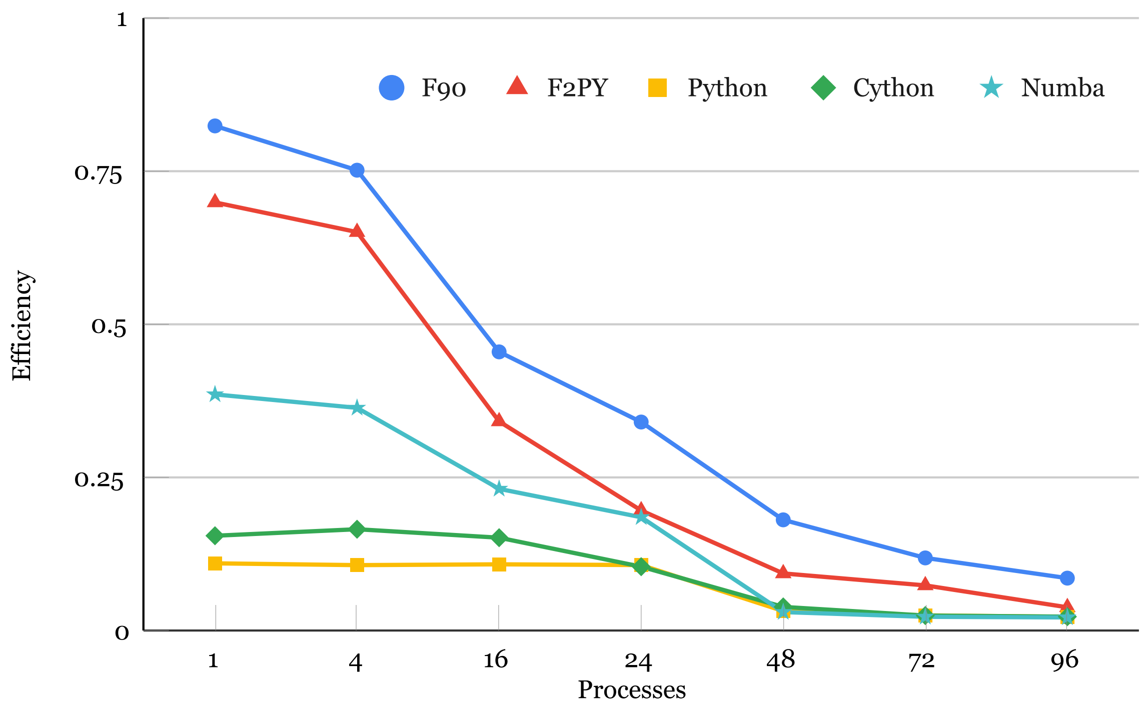
Source: Author's production.

implementations, however in the parallel execution the results had little variation, and for 1, 4, 36 and 49 processes, F2PY obtained a better time than F90. For 9 MPI processes there was an exception, the times of both F90 and F2PY were worse than the time obtained by Numba. The rest of the comments are very similar to the comments in the subsubsection 5.2.1.1. Additionally, in the case of 9 MPI processes, it would be interesting to investigate why Numba got the best time.

5.3.1.2 Comments on obtained F2PY serial and parallel performance

F2PY, together with F90, were the best performing implementations. Despite the overhead added by the wrapper and the Python interpreter, F2PY achieved superior performance for 1, 4, 36, and 49, and also in the serial implementation. For 9 processes, the performance of F2PY and F90 was worse than Numba. The rest of the comments are very similar to the comments in the second paragraph of the subsubsection 5.2.1.3.

Figure 5.9 - Parallel efficiency of FFT test case implementations, depending on the number of processes



Source: Author’s production.

5.3.1.3 Comments on obtained Numba serial and parallel performance

In general, Numba’s performance was below F90 and F2PY, but above Python and Cython implementations. The performance of the parallel implementation for 9 MPI processes was the best of the implementations, with 2.41 s, even better than F90 and F2PY. The rest of the comments are very similar to the comments in the subsubsection 5.2.1.5.

5.3.1.4 Standard Python serial and parallel performance

The comments for this implementation in standard Python are almost the same as those described in subsubsection 5.2.1.2, the execution time is high but the code serves as a reference because it runs without changes in different environments that have standard Python. Code snippets that are computationally intensive can, in a second step, be optimized using F2PY, Cython, Numba, or other resources, usually by creating a library module to replace the original snippet, or you can use one of the existing libraries, if applicable.

5.3.1.5 Comments on obtained Cython serial and parallel performance

In general, the performance of the Cython implementation is the worst, and the times were the worst in both the serial and parallel implementation. Cython in the serial version uses some Cython extensions, uses the pyFFTW library, and creates an external module that is called by the main code in standard Python. In the parallel version it uses the mpi4py-fft library, and the rest of the code is executed by the standard Python interpreter, which includes the use of the mpi4py library. In this implementation Cython is using an external library for calculation, instead of using an algorithm programmed in pure Cython. 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.

5.3.1.6 Numa

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5.3.1.7 Overhead

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5.3.2 Processing performance of the Numba-GPU implementation

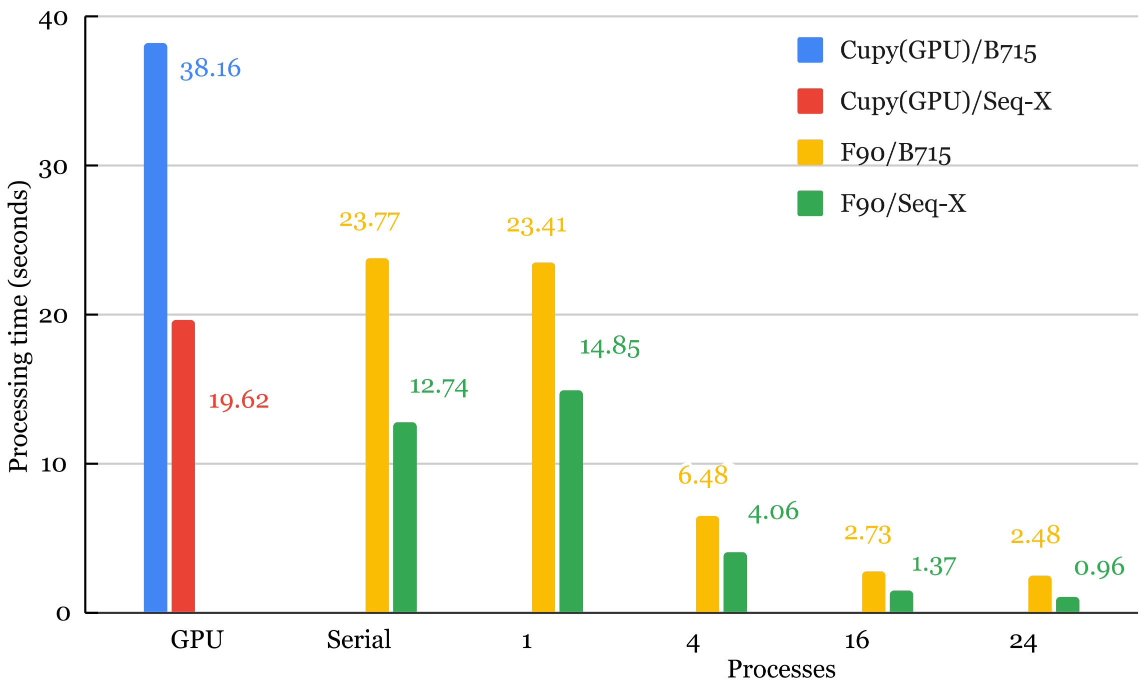
In a similar way to what happened at subsection 5.3.2, this section intends to compare the single-node performance of: (i) the single-process Numba-GPU implementation executed in a single-core and one GPU of the Bull B715 or the Bull Sequana-X node; (ii) the single-process F90 serial and parallel implementations 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 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.

Table 5.5 - Performance based on the number of processes for the F90 and Numba-GPU implementations performed on Bull Sequana-X and Bull B715. The execution time of the serial code compiled F90 in the Bull B715 node was taken as a reference for the calculation of speedup (highlighted in **blue**).

		Processing time (s)				
	GPU	**Serial**	**1**	**4**	**16**	**24**
F90/B715		**23.77**	23.41	6.48	2.73	2.48
F90/Seq-X		**12.74**	**14.85**	**4.06**	**1.37**	**0.96**
Cupy(GPU)/B715	38.16					
Cupy(GPU)/Seq-X	**19.62**					
		Speedup				
F90/B715		**1.00**	1.02	3.67	8.71	9.59
F90/Seq-X		**1.87**	**1.60**	**5.85**	**17.32**	**24.74**
Cupy(GPU)/B715	0.62					
Cupy(GPU)/Seq-X	**1.21**					
		Parallel efficiency				
F90/B715		**1.00**	1.02	0.92	0.54	0.40
F90/Seq-X		**1.87**	**1.60**	**1.46**	**1.08**	**1.03**
Cupy(GPU)/B715	0.62					
Cupy(GPU)/Seq-X	**1.21**					

Source: Author's production.

Figure 5.10 - Processing time (s) depending on the number of processes for the F90 and Numba-GPU implementations performed on the Bull Sequana-X (blue and purple bars) and on the Bull B715 node (red and green bars).



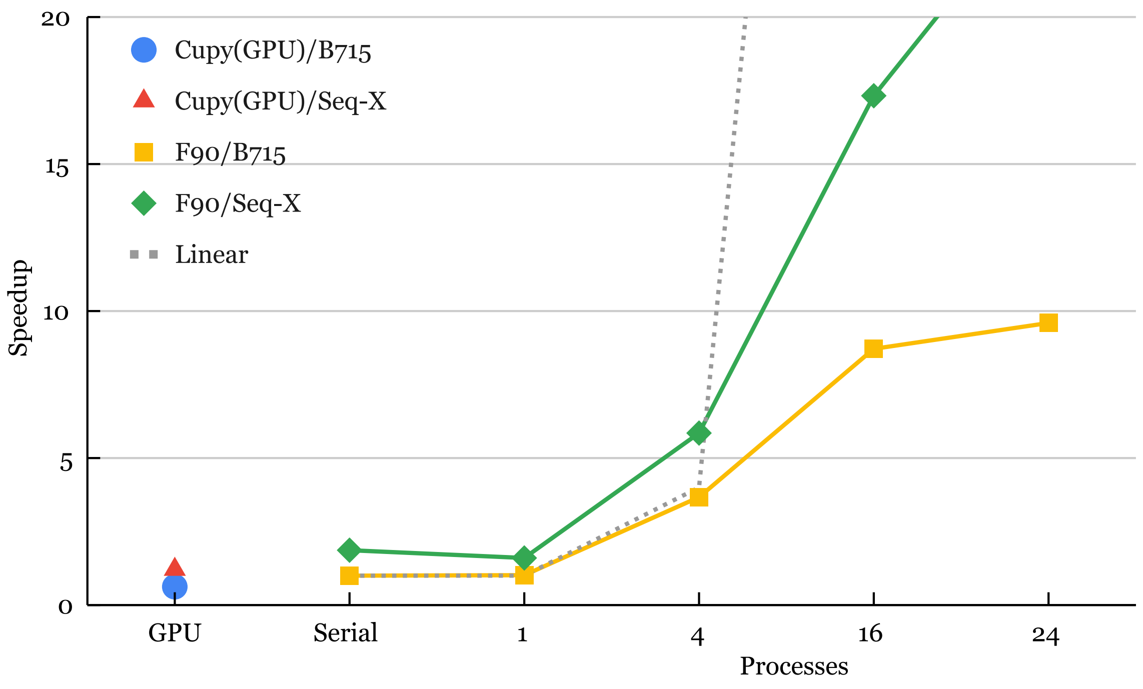
Source: Author's production.

The Numba-GPU implementation was tested in a Bull B715 node using a single core, and the Tesla K40 GPU runs in 17.21 s, close to the execution time of the serial F90 implementation (Figure 5.10). 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, a time that is close, but less than the F90 serial implementation on the same node . The runtime was longer than the time for all other parallel implementations.

The Numba-GPU implementation required more modifications in comparison to the standard serial Python code than the others. Similarly, the division, distribution of matrix, and calculation of the Fourier transform compose the main loop of the test case algorithm. The code was encapsulated in a function with a Numba hint for execution in a single GPU using just-in-time (JIT) compilation. The CuPy library was used, and the module for calculating the FFT was selected.

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 shown in Table 5.5 and in Figure 5.10, Figure 5.11, and

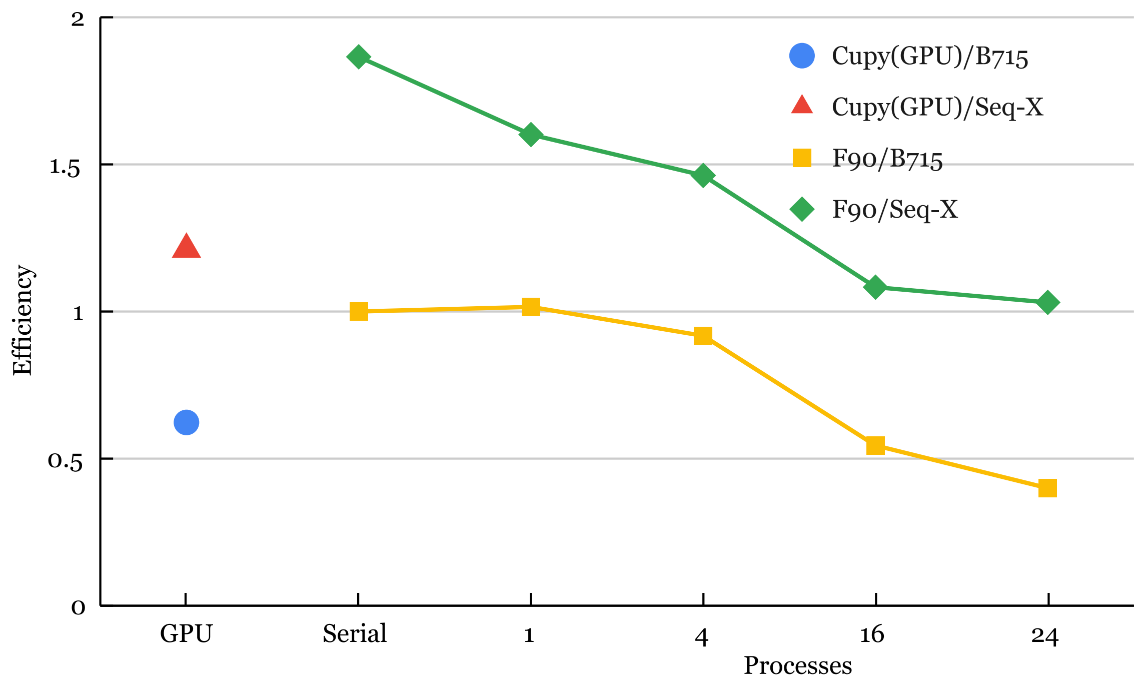
Figure 5.11 - Speedups depending on the number of processes for the F90 and Numba-GPU implementations executed on the Bull Sequana-X (blue triangle and purple line) and on the Bull B715 node (red point and green line). The execution time of the serial code compiled F90 in the Bull B715 node was taken as a reference for the calculation of speedup. The dashed line indicates the linear speedup.



Source: Author's production.

Figure 5.12. 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, in the same way as described in the subsection 5.3.2. 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.

Figure 5.12 - Parallel efficiencies depending on the number of processes for the F90 and Numba-GPU implementations performed on the Bull Sequana-X (blue triangle and purple line) and on the Bull B715 node (red dot and green line).



Source: Author’s production.

5.4 Analysis of the random forest test case

This section shows the processing performance of the random forest test case in much the same way as in the previous sections. During the analysis of the implementations, the repetition of the same texts that are in common will be avoided, but this will be indicated with the reference to the previous section. If desired, please read the reference provided to complete with more information.

5.4.1 Processing performance of implementations executed by the CPU

This subsection 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. The Table 5.6 shows processing times of the random forest 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, 16, 24, 48, 72, and 96 processes in the Bull B710 nodes. From 1 to 24 MPI processes, 1 execution node is used, 48 processes use 2 nodes, 72 uses 3, and 96 uses 4 execution nodes. Figure 5.13 also shows processing times in function of the number of MPI processes for the different implementations, and Figure 5.14 shows the corresponding speedups calculated taking the F90 serial time as reference.

According to Table 5.6, the shortest time obtained was 11.00 s with Python using the Scikit-learn library and 24 MPI processes. The worst time obtained was 141.71 s with parallel F2PY using 1 MPI process. Python, Numba, and Cython use the Scikit-learn library (section 3.15), and F90 and F2PY use the PARF library (section 3.13). Scikit-learn is a native library for Python, and PARF is an native F90 library written in F90.

Numba serial obtained a slightly better time than Python serial, however to standardize the analysis, the Python time (using Scikit-learn) was used as a reference for speedup and efficiency calculations. Serial implementations achieved slightly better time than parallel versions with 1 MPI process, as the parallel version adds the overhead of parallel execution mechanisms and libraries. In general, the Python implementation performed best in most cases, being only marginally surpassed by serial Numba. For 1 and 4 MPI processes, Python performed much better than F90 and F2PY. From 16 processes onwards, the performance gap narrowed and remained stable, keeping the same characteristic, with Python getting the best performance,

Table 5.6 - Performance of the different implementations of the random forest test case described, depending on the number of MPI processes: processing times (seconds), accelerations and parallel efficiencies. The best values for serial or for each number of MPI processes are highlighted in **red**. The execution time of the serial code compiled with GNU-Fortran was taken as a reference for the calculation of speedup (in **blue**).

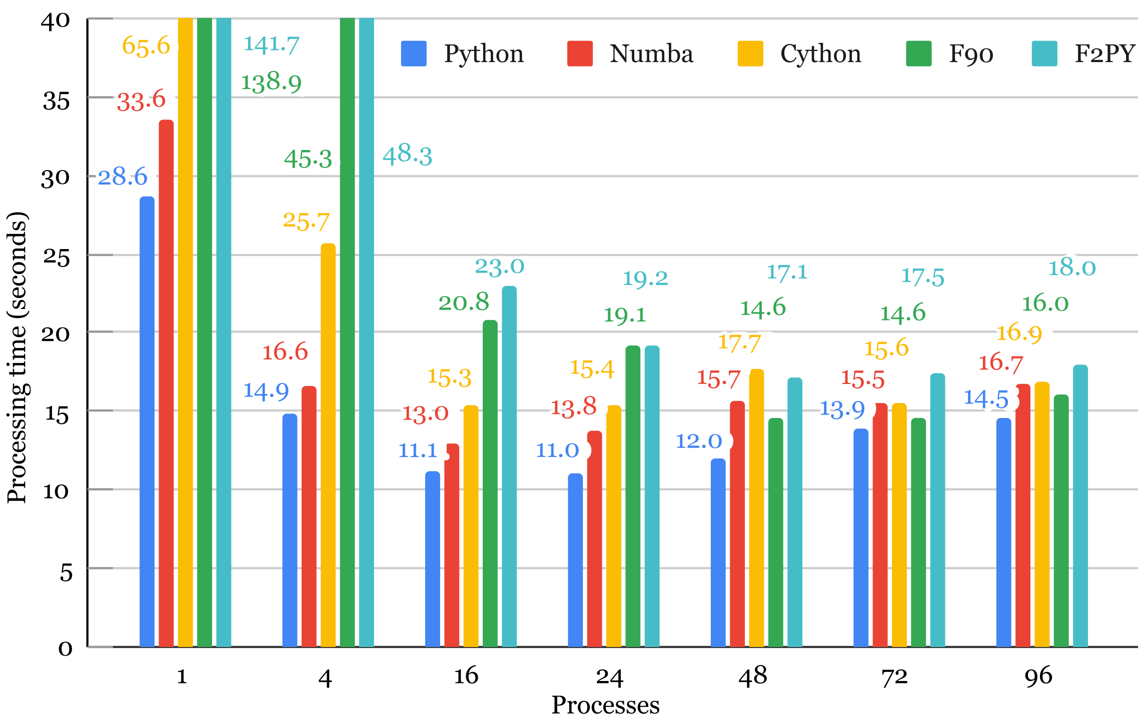
	Processing time (seconds)							
	Ser.	**1**	**4**	**16**	**24**	**48**	**72**	**96**
Python	**25.67**	**28.59**	**14.91**	**11.15**	**11.00**	**11.99**	**13.89**	**14.52**
Numba	**24.98**	33.61	16.63	13.01	13.80	15.65	15.51	16.69
Cython	29.46	65.56	25.71	15.32	15.41	17.71	15.58	16.89
F90	137.69	138.93	45.35	20.83	19.15	14.55	14.62	16.04
F2PY	135.10	141.71	48.29	23.00	19.16	17.14	17.46	17.96
	Speedup							
Python	**1.00**	**0.90**	**1.72**	**2.30**	**2.33**	**2.14**	**1.85**	**1.77**
Numba	**1.03**	0.76	1.54	1.97	1.86	1.64	1.65	1.54
Cython	0.87	0.39	1.00	1.67	1.67	1.45	1.65	1.52
F90	0.19	0.18	0.57	1.23	1.34	1.76	1.76	1.60
F2PY	0.19	0.18	0.53	1.12	1.34	1.50	1.47	1.43
	Parallel efficiency							
Python	**1.00**	**0.90**	**0.43**	**0.14**	**0.10**	**0.04**	**0.03**	**0.02**
Numba	**1.03**	0.76	0.39	0.12	0.08	0.03	0.02	0.02
Cython	0.87	0.39	0.25	0.10	0.07	0.03	0.02	0.02
F90	0.19	0.18	0.14	0.08	0.06	0.04	0.02	0.02
F2PY	0.19	0.18	0.13	0.07	0.06	0.03	0.02	0.01

Source: Author's production.

and F2PY getting the worst performance.

The F90 and F2PY implementations use the PARF library which is not as optimized as Scikit-learn, nor actively developed, and the results showed lower performance for the developed implementation. As most of the work is done by the used libraries, it was not necessary great coding efforts, which were concentrated in configuring parameters and dataset, and in the parallel case, establishing the parallelization environment. The low performance of the F90 and F2PY versions shows the convenience of using optimized and actively developed libraries in implementations. The results show that either Python or Numba or Cython are better options than F90 and F2PY, in the case of the implementations that were made. We could also have used the Scikit-learn

Figure 5.13 - Processing times for the different implementations of the random forest test case as a function of the number of processes (for convenience, times above 40 s do not appear in the graph)

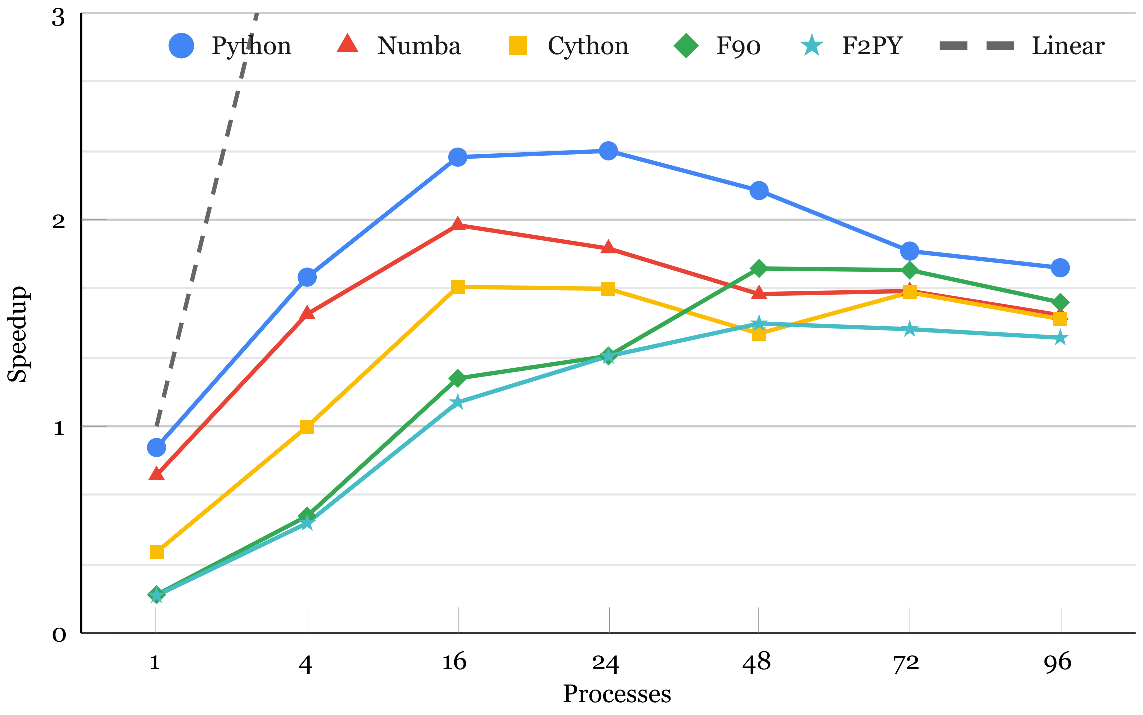


Source: Author's production.

library in F90, through an interface, and the times obtained would probably have been similar to Python's, but in this case the use of a native F90 library written in F90 served to compare different alternatives and possibly help in future decision-making. In the implementations it was also possible to see an example of Python working as a glue language, using the interface facility to work with an optimized library written in other languages, and connecting different components together to obtain performance. It is also important to note that the results obtained for the implementation of a random forest may not be the same for other implementations, in this specific case the Scikit-learn library behaved better, but in other implementations other libraries might not obtain the same result.

Numba and Cython, despite being compilers that translate to machine language, the former at runtime (JIT) and the latter before execution (AOT), did not get a performance boost over Python. This shows that for the implementations performed, it is more important to use an optimized library than a compiler. A particularly

Figure 5.14 - Speedups of random forest test case implementations as a function of the number of processes. Serial Python time was used as reference for calculations. The dotted line denotes linear speedup

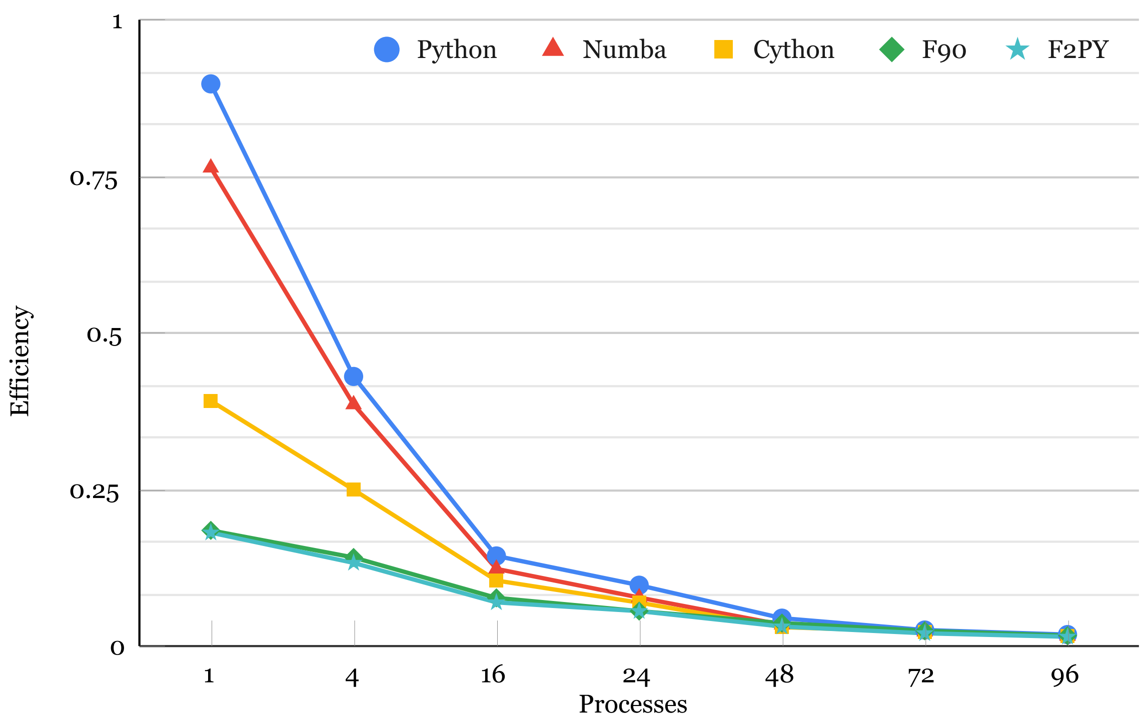


Source: Author's production.

interesting result is that interpreted Python (using Scikit-learn) was faster than compilers, due to the computational effort being made by the optimized library.

The reference time to calculate speedup was the same Python serial time for all implementations. Rankings for speedups and parallel efficiencies are similar for processing times. As can be seen in Table 5.6, Figure 5.13 and Figure 5.14, the parallel scalability grows from 1 to 16 MPI processes (1 execution node). In the case of F90 and F2PY the scalability grows up to 48 MPI processes (2 nodes). After that, speedup either stabilizes or decreases, and all implementations tend to a speedup value between 1 and 2. In the case of Python, Numa, and Cython, which use Scikit-learn, when using more than 1 node (above of 24 processes), efficiency decreases. At first glance it might seem that this is due to the MPI communication required between processes in different execution nodes, but in the case of F90 and F2PY using 48 MPI processes, two execution nodes are being used and speedup is growing, showing that there are others factors involved to justify the speedup drop.

Figure 5.15 - Parallel efficiencies of random forest test case implementations as a function of number of processes



Source: Author's production.

It would be interesting to investigate these factors in future work.

For Python, up to 16 processes the speedup curve kept growing approximately linearly. Between 16 and 24 processes there was no great increase in speedup. It is interesting to note that between 16 and 24 processes, the execution takes place in a single execution node, and therefore the MPI communication time at first should continue to grow, showing that the problem of efficiency drop also occurs within a node of execution. This is most noticeable for Numba and Cython, who had a speedup decrease within an execution node. Interestingly, the same doesn't happen for F90 and F2PY between 16 and 48 processes, where speedup keeps growing, in contrast to Python. After 48 processes, the behavior for all implementations is approximately the same. In general, the speedup obtained is relatively low, even between 1 and 16 processes, being far from the dotted curve showing the linear speedup.

Parallel efficiency, for all implementations, tends to zero as the number of MPI processes increases, and all implementations follow the same behavior. Python has a higher parallel efficiency for a small number of processes, and F90 and P2PY have low efficiency, but as the number of processes increases, the efficiencies get closer, and tending to zero, showing a low scalability for the developed implementations.

5.4.1.1 Comments on obtained F90 serial and parallel performance

The F90 code requires the PARF library to be configured and compiled using a compatible compiler, and it also needs to be configured to use the MPI library to allow for parallelization. PARF has an interface to the command line, which was used in the implementation. Two libraries are compiled and generated, one for the sequential version, the other for the parallel version, the parallel including the MPI library during compilation. Once created, the libraries are used directly via the command line, using the files containing the datasets as parameters. In the case of the parallel version there is no need for major changes to the Slurm commit files. The F90 version in general follows the same efficiency and speedup behavior as the other versions, with between 16 and 48 processes (1 and 2 execution nodes) the speedup behavior is different, with speedup increasing, while Python, Numba, and Cython tended to stabilize or decrease. F90 and F2PY had the worst execution times, but the time differences between them and Python decrease as the number of processes increases. The acceleration of implementations tends to a close value, between 1.4 and 1.8, as the number of MPI processes increases. Running F90 and F2PY on SDumont also followed the traditional F90 model, compiling, using the MPI library, and the Slurm workload manager, in contrast to the need to use ipython parallel engines of the Scikit-learn versions.

5.4.1.2 Comments on obtained F2PY serial and parallel performance

For the F2PY implementation, the F90 code (PARF library) was changed to be able to be converted to a Python library using F2PY which in turn is a NumPy resource to reuse existing F90 code. The newly created Python library is then used in the implementation, just like an ordinary Python library. The Python part of the implementation is a relatively short code that loads the library, loads the datasets, calls the library, and displays the result. Overall F2PY performance is close to F90 but the overhead layer adds a little extra time. In the case of the serial F2PY version it is slightly faster than the F90 version, however the F90 code has been changed and recompiled, and this could be the reason.

5.4.1.3 Comments on obtained Numba serial and parallel performance

The Numba implementation uses JIT compilation, however the computational effort is also made by the Scikit-learn library. The overhead layer added by runtime compilation makes the runtime slightly longer than Cython, which in turn is longer than Python.

5.4.1.4 Standard Python serial and parallel performance

The Python implementation uses the Scikit-learn library that performs the computationally intensive part. Standard Python code reads the datasets and performs some operations to adapt and convert the data to the format that the library accepts. Then the library parameters are set, the library is called, and finally the result is displayed. The implementation is relatively simple, however standard Python code is more complex than the F2PY implementation, considering that the F90 code exists and has been reused. The code implementation effort is less in Python (with Scikit-learn) than F90, and in this case Python is faster.

5.4.1.5 Comments on obtained Cython serial and parallel performance

The Cython implementation has few changes compared to Python, but adds a layer of overhead that increases execution time a bit. This implementation did not use all Cython resources, however considering that the computational effort continues to be performed by the Scikit-learn library, it is not expected a great performance increase when using more Cython resources. The results show that for the implementations made, standard Python is the best option, both in terms of performance and in code development effort.

5.4.1.6 Overhead

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5.5 Profiling

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5.5.1 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

5.5.1.1 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus

vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

5.5.1.2 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5.5.1.3 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5.5.1.4 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5.5.1.5 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5.5.2 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus

vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

6 FINAL REMARKS

This work presented a comparison of the most common HPC approaches in Python applied to three test cases and executed in the LNCC Santos Dumont supercomputer:

- Finite difference method to solve the partial differential equations resulting from the Poisson equations for a 2D heat transfer problem. It requires the calculation of a 5-point stencil over the discretized domain grid;
- Fast Fourier transform, used for example in signal and image processing, in which the three-dimensional domain is decomposed, along one dimension, into slices along the chosen dimension;
- Random decision forest, used for example for classification and regression tasks, is a machine learning method that operates by creating randomly a combination of decision trees, and in general uses cases trained by a classifier method.

Their serial and parallel implementations in F90 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 approach 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.

6.1 Future work

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

REFERENCES

ABRAHAMS, D.; GROSSE-KUNSTLEVE, R. W. Building Hybrid Systems with Boost.Python. **C/C++ Users Journal**, v. 21, n. 7, 5 2003. OSTI 815409. 25

AGIUS, I.-M.; INGUANEZ, F. Re-purposing Computers as a Cluster for Academic Institutions. In: IEEE. **2019 IEEE 9th International Conference on Consumer Electronics (ICCE-Berlin)**. [S.l.], 2019. p. 57–60. DOI 10.1109/ICCE-Berlin47944.2019.8966155. 22

ALNÆS, M.; BLECHTA, J.; HAKE, J.; JOHANSSON, A.; KEHLET, B.; LOGG, A.; RICHARDSON, C.; RING, J.; ROGNES, M. E.; WELLS, G. N. The FEniCS Project Version 1.5. **Archive of Numerical Software**, v. 3, n. 100, 2015. DOI 10.11588/ans.2015.100.20553. 22

BALAJI, P.; GROPP, W.; HOEFLER, T.; THAKUR, R. Advanced MPI Programming. In: **Tutorial at SC17: The International Conference for High Performance Computing, Networking, Storage, and Analysis**. Argonne National Laboratory, 2017. P. Balaji (Argonne National Laboratory), W. Gropp (Universityof Illinois, Urbana-Champaign), T. Hoefler (ETH Zurich), R. Thakur (Argonne National Laboratory). Available from: <`https://www.mcs.anl.gov/~thakur/sc17-mpi-tutorial/`>. Access in: Dec 2021. 29, 113

BARNEY, B. Message Passing Interface (MPI). **Lawrence Livermore National Laboratory**, 2021. LLNL HPC Tutorials. UCRL-MI-133316. Available from: <`http://computing.llnl.gov/tutorials/mpi/`>. Access in: Dec 2021. 2

BEAZLEY, D. M.; LOMDAHL, P. S. Feeding a Large-scale Physics Application to Python. In: **Proceedings of the 6th International Python Conference**. [s.n.], 1997. OSTI 658327. Available from: <`http://legacy.python.org/workshops/1997-10/proceedings/beazley.html`>. Access in: Dec 2021. 2

BEHNEL, S.; BRADSHAW, R.; CITRO, C.; DALCIN, L.; SELJEBOTN, D. S.; SMITH, K. Cython: The Best of Both Worlds. **Computing in Science & Engineering**, IEEE, v. 13, n. 2, p. 31–39, 2010. DOI 10.1109/MCSE.2010.118. 8, 12, 13, 32

BERGSTRA, J.; BREULEUX, O.; BASTIEN, F.; LAMBLIN, P.; PASCANU, R.; DESJARDINS, G.; TURIAN, J.; WARDE-FARLEY, D.; BENGIO, Y. Theano: A

CPU and GPU Math Compiler in Python. In: **Proceedings of the 9th Python in Science Conference (SciPy 2010)**. [S.l.: s.n.], 2010. v. 1, p. 18 – 24. DOI 10.25080/Majora-92bf1922-003. 22

BIHAM, E.; SEBERRY, J. Pypy: Another Version of Py. **eSTREAM, ECRYPT Stream Cipher Project, Report 2006/038**, v. 38, p. 2006, 2006. Available from: <http://www.ecrypt.eu.org/stream/pyp2.html>. Access in: Dec 2021. 7, 24

BISONG, E. **JupyterLab Notebooks**. Berkeley, CA: Apress, 2019. 49–57 p. DOI 10.1007/978-1-4842-4470-8_6. ISBN 978-1-4842-4470-8. 9

BLANK, D.; KUMAR, D.; MEEDEN, L.; YANCO, H. Pyro: A Python-based Versatile Programming Environment for Teaching Robotics. **Journal on Educational Resources in Computing (JERIC)**, ACM New York, NY, USA, v. 3, n. 4, p. 1–es, 2003. DOI 10.1145/1047568.1047569. 24

BOULESTEIX, A.-L.; JANITZA, S.; KRUPPA, J.; KÖNIG, I. R. Overview of random forest methodology and practical guidance with emphasis on computational biology and bioinformatics. **Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery**, Wiley Online Library, v. 2, n. 6, p. 493–507, 2012. 19

BROWNLEE, J. **Deep Learning with Python: Develop Deep Learning Models on Theano and TensorFlow Using Keras**. [S.l.]: Machine Learning Mastery, 2016. 22

CHEN, C. J.; BERNATZ, R. A.; CARLSON, K. D.; LIN, W.; DAVIS, G. V. de. Finite Analytic Method in Flows and Heat Transfer. **Applied Mechanics Reviews**, ASME International, v. 55, n. 2, p. B34–B34, 2002. DOI 10.1115/1.1451171. 27

CHOI, J.; FINK, Z.; WHITE, S.; BHAT, N.; RICHARDS, D. F.; KALE, L. V. GPU-aware Communication with UCX in Parallel Programming Models: Charm++, MPI, and Python. **arXiv**, 2021. ArXiv:2102.12416 [cs.DC]. 23

CIELO, S.; IAPICHINO, L.; BARUFFA, F. Speeding Simulation Analysis up with yt and Intel Distribution for Python. **arXiv**, 2019. ArXiv:1910.07855 [astro-ph.IM]. 1, 22

COLLOBERT, R.; BENGIO, S.; MARIÉTHOZ, J. **Torch: a Modular Machine Learning Software Library**. [S.l.], 2002. Idiap-RR-46-2002. Available from: <http://publications.idiap.ch/index.php/publications/show/712>. Access in: Dec 2021. 14

DALCÍN, L.; PAZ, R.; STORTI, M.; D'ELÍA, J. MPI for Python: Performance Improvements and MPI-2 Extensions. **Journal of Parallel and Distributed Computing**, Elsevier BV, v. 68, n. 5, p. 655?662, 2008. ISSN 0743-7315. DOI 10.1016/j.jpdc.2007.09.005. 8, 12, 30

DANIEL, T. R.; MIRCEA, S. AES on GPU using CUDA. In: **2010 European Conference for the Applied Mathematics & Informatics. World Scientific and Engineering Academy and Society Press**. [S.l.: s.n.], 2010. p. 225 – 230. ISBN 978-960-474-260-8. ISSN 1792-7390. 35

De Rainville, F.-M.; FORTIN, F.-A.; GARDNER, M.-A.; PARIZEAU, M.; GAGNÉ, C. DEAP: A Python Framework for Evolutionary Algorithms. In: **Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation**. [S.l.: s.n.], 2012. p. 85–92. DOI 10.1145/2330784.2330799. 23

DONGARRA, J. J.; OTTO, S. W.; SNIR, M.; WALKER, D. **An Introduction to the MPI Standard**. USA, 1995. DOI 10.5555/898812. 30

DRABAS, T.; LEE, D. **Learning PySpark**. [S.l.]: Packt Publishing Ltd, 2017. 24

FAOUZI, J.; JANATI, H. pyts: A Python Package for Time Series Classification. **Journal of Machine Learning Research**, v. 21, n. 46, p. 1–6, 2020. Available from: <http://jmlr.org/papers/v21/19-763.html>. Access in: Dec 2021. 21

FARRELL, S.; VOSE, A.; EVANS, O.; HENDERSON, M.; CHOLIA, S.; PÉREZ, F.; BHIMJI, W.; CANON, S.; THOMAS, R.; PRABHAT, M. Interactive Distributed Deep Learning with Jupyter Notebooks. In: _____. [S.l.]: ISC High Performance 2018 International Workshops, Frankfurt/Main, Germany, June 28, 2018, Revised Selected Papers, 2018. p. 678–687. ISBN 978-3-030-02464-2. DOI 10.1007/978-3-030-02465-9_49. 11

FEY, M.; LENSSEN, J. E. Fast Graph Representation Learning with PyTorch Geometric. **CoRR**, 2019. ArXiv:1903.02428 [cs.LG]. 14

FRIGO, M.; JOHNSON, S. G. FFTW: An adaptive software architecture for the FFT. In: IEEE. **Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)**. [S.l.], 1998. v. 3, p. 1381–1384. 18

GAYER, A. V.; CHERNYSHOVA, Y. S.; SHESHKUS, A. V. Effective Real-time Augmentation of Training Dataset for the Neural Networks Learning. In: INTERNATIONAL SOCIETY FOR OPTICS AND PHOTONICS. **Proceedings**

of the 11th International Conference on Machine Vision (ICMV 2018). [S.l.]: SPIE, 2019. v. 11041, p. 394 – 401. DOI 10.1117/12.2522969. 22

GOMERSALL, H. **pyFFTW: Python wrapper around FFTW**. sep. 2021. ascl:2109.009 p. Provided by the SAO/NASA Astrophysics Data System. Available from: <https://ui.adsabs.harvard.edu/abs/2021ascl.soft09009G>. Access in: Dec 2021. 19

GROPP, W.; LUSK, E. User's Guide for MPICH, a Portable Implementation of MPI. **Mathematics and Computer Science Division, Argonne National Laboratory**, jul 1996. OSTI 378910. DOI 10.2172/378910. 12, 30

GROPP, W.; LUSK, E.; DOSS, N.; SKJELLUM, A. A High-performance, Portable Implementation of the MPI Message Passing Interface Standard. **Parallel Computing**, v. 22, n. 6, p. 789–828, 1996. ISSN 0167-8191. DOI 10.1016/0167-8191(96)00024-5. 2

GRÜNING, B.; DALE, R.; SJÖDIN, A.; CHAPMAN, B. A.; ROWE, J.; TOMKINS-TINCH, C. H.; VALIERIS, R.; KÖSTER, J. Bioconda: sustainable and comprehensive software distribution for the life sciences. **Nature methods**, Nature Publishing Group, v. 15, n. 7, p. 475–476, 2018. 9

GUELTON, S. Pythran: Crossing the Python Frontier. **Computing in Science & Engineering**, IEEE, v. 20, n. 2, p. 83–89, 2018. DOI 10.1109/MCSE.2018.021651342. 25

HINSEN, K. The Molecular Modeling Toolkit: A Case Study of a Large Scientific Application in Python. In: **Proceedings of the 6th International Python Conference**. [s.n.], 1997. Available from: <http://legacy.python.org/workshops/1997-10/proceedings/hinsen.html>. Access in: Dec 2021. 2

HOLD-GEOFFROY, Y.; GAGNON, O.; PARIZEAU, M. Once you SCOOP, no Need to Fork. In: **Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment**. [S.l.: s.n.], 2014. p. 1–8. DOI 10.1145/2616498.2616565. 24

HOLM, H. H.; BRODTKORB, A. R.; SÆTRA, M. L. GPU Computing with Python: Performance, Energy Efficiency and Usability. **Computation**, Multidisciplinary Digital Publishing Institute, v. 8, n. 1, p. 4, 2020. DOI 10.3390/computation8010004. 22

IEEE Spectrum. **Interactive: The Top Programming Languages 2020**. 2020. Available from: <http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020>. Access in: Dec 2021. 1

INPE. **URLib: A Platform for a Digital Library or Archive Distributed on the Web Used by the Brazilian National Institute for Space Research (INPE)**. 2020. Available from: <http://www.urlib.net>. Access in: Dec 2021. 1

KETKAR, N. Introduction to Pytorch. In: **Deep Learning With Python**. [S.l.]: Apress, Berkeley, CA, 2017. p. 195–208. DOI 10.1007/978-1-4842-2766-4_12. 8, 13, 14

KLÖCKNER, A.; PINTO, N.; CATANZARO, B.; LEE, Y.; IVANOV, P.; FASIH, A. Chapter 27 - GPU Scripting and Code Generation with PyCUDA. In: HWU, W.-m. W. (Ed.). **GPU Computing Gems Jade Edition**. Boston: Morgan Kaufmann, 2012, (Applications of GPU Computing Series). p. 373–385. ISBN 978-0-12-385963-1. DOI 10.1016/B978-0-12-385963-1.00027-7. 17

KLÖCKNER, A.; PINTO, N.; LEE, Y.; CATANZARO, B.; IVANOV, P.; FASIH, A. PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation. **Parallel Computing**, Elsevier BV, v. 38, n. 3, p. 157–174, 2012. DOI 10.1016/j.parco.2011.09.001. 17, 26

LAM, S. K.; PITROU, A.; SEIBERT, S. Numba: A LLVM-Based Python JIT Compiler. In: **Proceedings of the 2nd Workshop on the LLVM Compiler Infrastructure in HPC**. New York, NY, USA: Association for Computing Machinery, 2015. (LLVM '15). ISBN 9781450340052. DOI 10.1145/2833157.2833162. 7, 8, 13, 34

LAM, S. K.; SEIBERT, S. How to Accelerate an Existing Codebase with Numba. In: **SciPy 2019, the 18th annual Scientific Computing with Python conference**. SciPy, 2019. Available from: <http://www.scipy2019.scipy.org/talks-posters/How-to-Accelerate-an-Existing-Codebase-with-Numba>. Access in: Dec 2021. 14

LANGTANGEN, H. P.; CAI, X. On the Efficiency of Python for High-Performance computing: A Case Study Involving Stencil Updates for Partial Differential equations. In: BOCK, H. G.; KOSTINA, E.; PHU, H. X.; RANNACHER, R. (Ed.). **Modeling, Simulation and Optimization of Complex Processes**. [S.l.]: Springer, 2008. p. 337?357. ISBN 978-3-540-79409-7. DOI 10.1007/978-3-540-79409-7_23. 28

LATTNER, C.; ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: IEEE. **International Symposium on Code Generation and Optimization, 2004. CGO 2004.** [S.l.], 2004. p. 75–86. DOI 10.1109/CGO.2004.1281665. 13

LI, Y.; SUN, H.; YUAN, J.; WU, T.; TIAN, Y.; LI, Y.; ZHOU, R.; LI, K.-C. **Accelerating 3D Digital Differential Analyzer Ray Tracing Algorithm on the GPU Using CUDA**. 2015. DOI 10.1109/icppw.2015.44. 36

LUNACEK, M.; BRADEN, J.; HAUSER, T. The scaling of many-task computing approaches in python on cluster supercomputers. In: IEEE. **2013 IEEE International Conference on Cluster Computing (CLUSTER)**. [S.l.], 2013. p. 1–8. DOI 110.1109/CLUSTER.2013.6702678. 1

MAROWKA, A. On Parallel Software Engineering Education Using Python. **Education and Information Technologies**, Springer, v. 23, n. 1, p. 357–372, 2018. DOI 10.1007/s10639-017-9607-0. 13

______. Python Accelerators for High-performance Computing. **The Journal of Supercomputing**, Springer, v. 74, n. 4, p. 1449–1460, 2018. DOI 10.1007/s11227-017-2213-5. 25

MCKINNEY, W. et al. pandas: a foundational python library for data analysis and statistics. **Python for high performance and scientific computing**, Dallas, TX, v. 14, n. 9, p. 1–9, 2011. 20

MCLEOD, C. A Framework for Distributed Deep Learning Layer Design in Python. **CoRR**, 2015. ArXiv:1510.07303 [cs.LG]. 23

MIRANDA, E. F.; STEPHANY, S. Common Approaches to HPC in Python Evaluated for a Scientific Computing Test Case. **Revista Cereus**, v. 13, n. 2, p. 84–98, 2021. ISSN 2175-7275. DOI 10.18605/2175-7275/cereus.v13n2p84-98. ORCID Miranda, E. F. 0000-0003-1200-794X. ORCID Stephany, S. 0000-0002-6302-4259. 3

______. Comparison of High Performance Computing Approaches in the Python Environment for a Five-Point Stencil Test Problem. In: **XV Brazilian e-Science Workshop (BreSci), event that is part of XLI Congress of the Brazilian Computer Society (CSBC-2021), from 18 to 23 July 2021**. [S.l.: s.n.], 2021. p. 33–40. ISSN 2763-8774. DOI 10.5753/bresci.2021.15786. ORCID Miranda, E. F. 0000-0003-1200-794X. ORCID Stephany, S. 0000-0002-6302-4259. 3

MORITZ, P.; NISHIHARA, R.; WANG, S.; TUMANOV, A.; LIAW, R.; LIANG, E.; ELIBOL, M.; YANG, Z.; PAUL, W.; JORDAN, M. I. et al. Ray: A Distributed Framework for Emerging AI Applications. In: **13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)**. Carlsbad, CA: USENIX Association, 2018. p. 561–577. ISBN 978-1-939133-08-3. Available from: <http://www.usenix.org/conference/osdi18/presentation/moritz>. Access in: Dec 2021. 23

MORTENSEN, M.; DALCIN, L.; KEYES, D. E. mpi4py-fft: Parallel fast fourier transforms with mpi for python. **Journal of Open Source Software**, v. 4, n. 36, p. 1340, 2019. DOI 10.21105/joss.01340. 18

NASCIMENTO, F. J. B. D.; FILHO, L. R. A.; GUIMARÃES, L. N. F. CINTIA 2: A Hierarchy of Binary Artificial Neural Networks for the Intelligent Classification of Supernovae (in Portuguese). **Brazilian Journal of Applied Computing**, UPF Editora, v. 11, n. 2, p. 31–41, may 2019. DOI 10.5335/rbca.v11i2.9037. 1

NISHINO, R.; LOOMIS, S. H. C. CuPy: A NumPy-compatible Library for NVIDIA GPU Calculations. **NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems**, p. 151, 2017. 18

ODEN, L. Lessons Learned from Comparing C-CUDA and Python-Numba for GPU-Computing. In: IEEE. **2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)**. [S.l.], 2020. p. 216–223. DOI 10.1109/PDP50117.2020.00041. 22

OLIPHANT, T. E. Python for Scientific Computing. **Computing in Science & Engineering**, IEEE, v. 9, n. 3, p. 10–20, 2007. DOI 10.1109/MCSE.2007.58. 25

OSMIALOWSKI, P. How the Flang Frontend Works: Introduction to the Interior of the Open-source Fortran Frontend for LLVM. In: **Proceedings of the 4th Workshop on the LLVM Compiler Infrastructure in HPC**. [S.l.: s.n.], 2017. p. 1–14. DOI 10.1145/3148173.3148183. 22

PALACH, J. **Parallel Programming with Python**. [S.l.]: Packt Publishing Ltd, 2014. 8, 23

PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V. et al. Scikit-learn: Machine learning in python. **the Journal of machine Learning research**, JMLR. org, v. 12, p. 2825–2830, 2011. 20

PERKEL, J. M. Why Jupyter is Data Scientists' Computational Notebook of Choice. **Nature**, Nature Publishing Group, v. 563, n. 7732, p. 145–147, 2018. DOI 10.1038/d41586-018-07196-1. 9

PETERSON, P. F2Py: A Tool for Connecting Fortran and Python Programs. **International Journal of Computational Science and Engineering**, Inderscience Publishers, v. 4, n. 4, p. 296–305, 2009. DOI 10.1504/IJCSE.2009.029165. 8, 17, 32

ROCKLIN, M. Dask: Parallel Computation with Blocked Algorithms and Task Scheduling. In: **Proceedings of the 14th Python in Science Conference**. [S.l.: s.n.], 2015. v. 126, p. 126–132. DOI 10.25080/Majora-7b98e3ed-013. 8, 15

_____. **Dask, Advanced Parallelism for Analytics**. 2021. Open source library for parallel computing written in Python, originally developed by Matthew Rocklin, a community project maintained and sponsored by developers and organizations, and several projects use it in an integrated way to feed components of their infrastructure. Available from: <http://dask.org/>. Access in: Dec 2021. 15

SANNER, M. Python: A Programming Language for Software Integration and Development. **Journal of Molecular Graphics and Modelling**, v. 17, n. 1, p. 57–61, 1999. Available from: <http://pubmed.ncbi.nlm.nih.gov/10660911/>. Access in: Dec 2021. 5, 6

SEBESTA, R. W.; MUKHERJEE, S.; BHATTACHARJEE, A. K. **Concepts of Programming Languages**. [S.l.]: Boston: Pearson, 2016. ISBN 9780133943023. 5

SEHRISH, S.; KOWALKOWSKI, J.; PATERNO, M.; GREEN, C. Python and HPC for High Energy Physics Data Analyses. In: **Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing**. New York, NY, USA: Association for Computing Machinery, 2017. (PyHPC'17). ISBN 9781450351249. DOI 10.1145/3149869.3149877. 1

SINGH, N.; BROWNE, L.-M.; BUTLER, R. Parallel Astronomical data Processing with Python: Recipes for Multicore Machines. **Astronomy and Computing**, Elsevier, v. 2, p. 1–10, 2013. DOI 10.1016/j.ascom.2013.04.002. 24

SINGH, S. Declarative Data-parallel Programming with the Accelerator System. In: **Proceedings of the 5th ACM Sigplan Workshop on Declarative Aspects of Multicore Programming**. [S.l.: s.n.], 2010. p. 1–2. DOI 10.1145/1708046.1708048. 5

SOUZA, C. R. d.; PANETTA, J.; STEPHANY, S. Analysis of Communication Performance Using MPI 3.0 Shared Memory Functionality (in Portuguese). **Revista Cereus**, Revista Cereus, v. 10, n. 2, p. 193–210, 2018. ISSN 2175-7275. DOI 10.18605/2175-7275/cereus.v10n2p193-210. 30

SOUZA, J.; MENDES, C.; SANTOS, R. Performance Optimization of the Brazil Solar Radiation Model (in Portuguese). In: **Proceedings of ERAD-2018 São Paulo IX Regional School of HPC (in Portuguese)**. [S.l.]: Brazilian Computer Society (SBC), 2018. DOI 10.5753/eradsp.2018.13601. 1

University of Tennessee; Los Alamos National Laboratory; Indiana University; University of Stuttgart. **OpenMPI: An Open Source Message Passing Interface Implementation**. dec. 2020. Message Passing Interface (MPI) library combining technologies and features from several other projects, used by many TOP500 supercomputers, including Roadrunner, and K computer. Available from: <`http://www.open-mpi.org/`>. Access in: Dec 2021. 30

Van Rossum, G. et al. Python Programming Language. In: **2007 USENIX Annual Technical Conference**. [s.n.], 2007. v. 41, p. 36. Available from: <`http://www.usenix.org/conference/2007-usenix-annual-technical-conference/presentation/python-programming-language`>. Access in: Dec 2021. 25

VASILEV, V.; CANAL, P.; NAUMANN, A.; RUSSO, P. Cling – the new Interactive Interpreter for ROOT 6. In: IOP PUBLISHING. **Journal of Physics: Conference Series**. [S.l.], 2012. p. 052071. DOI 10.1088/1742-6596/396/5/052071. 6

VIRTANEN, P.; GOMMERS, R.; OLIPHANT, T. E.; HABERLAND, M.; REDDY, T.; COURNAPEAU, D.; BUROVSKI, E.; PETERSON, P.; WECKESSER, W.; BRIGHT, J.; WALT, S. J. van der; BRETT, M.; WILSON, J.; MILLMAN, K. J.; MAYOROV, N.; NELSON, A. R. J.; JONES, E.; KERN, R.; LARSON, E.; CAREY, C. J.; POLAT, İ.; FENG, Y.; MOORE, E. W.; VANDERPLAS, J.; LAXALDE, D.; PERKTOLD, J.; CIMRMAN, R.; HENRIKSEN, I.; QUINTERO, E. A.; HARRIS, C. R.; ARCHIBALD, A. M.; RIBEIRO, A. H.; PEDREGOSA, F.; MULBREGT, P. van. Scipy 1.0: Fundamental Algorithms for Scientific Computing in Python. **Nature Methods**, Springer Science and Business Media LLC, v. 17, n. 3, p. 261–272, feb. 2020. DOI 10.1038/s41592-019-0686-2. 1

VIRTANEN, P.; GOMMERS, R.; OLIPHANT, T. E.; HABERLAND, M.; REDDY, T.; COURNAPEAU, D.; BUROVSKI, E.; PETERSON, P.; WECKESSER, W.;

BRIGHT, J. et al. Scipy 1.0: fundamental algorithms for scientific computing in python. **Nature methods**, Nature Publishing Group, v. 17, n. 3, p. 261–272, 2020. 21

WALT, S. V. D.; COLBERT, S. C.; VAROQUAUX, G. The NumPy Array: A Structure for Efficient Numerical Computation. **Computing in Science & Engineering**, IEEE, v. 13, n. 2, p. 22–30, 2011. DOI 10.1109/MCSE.2011.37. 8, 12, 13, 16, 31

YEAGER, L.; BERNAUER, J.; GRAY, A.; HOUSTON, M. Digits: the Deep Learning GPU Training System. In: **ICML 2015 AutoML Workshop**. [s.n.], 2015. Available from: <http://indico.ijclab.in2p3.fr/event/2914/contributions/6476/subcontributions/170>. Access in: Dec 2021. 22

APPENDIX A - PUBLISHED ARTICLES

Two different articles were submitted and approved for publication, one in a national congress and the other in a journal, and both cover only the five-point stencil case study:

- MIRANDA, E. F.; STEPHANY, S. *Comparison of High Performance Computing Approaches in the Python Environment for a Five-Point Stencil Test Problem*. In proceedings: XV Brazilian e-Science Workshop (BreSci), 2021. p. 33-40. ISSN 2763-8774. DOI 10.5753/bresci.2021.15786.

  Article approved and presented at the national congress (article 213763), XV Brazilian e-Science Workshop (BreSci), event that is part of the XLI Congress of the Brazilian Computer Society (CSBC-2021), from 18 to 23 July 2021. Available from: http://doi.org/10.5753/bresci.2021.15786. Access in: Dec. 2021.

- MIRANDA, E. F.; STEPHANY, S. *Common approaches to HPC in Python evaluated for a scientific computing test case*. REVISTA CEREUS, 13(2), 84-98, 2021. ISSN 2175-7275. DOI 10.18605/2175-7275/cereus.v13n2p84-98.

  Article accepted in a journal (submission 3408), REVISTA CEREUS. Qualis CAPES Interdisciplinary B2 in the evaluation of the 2013-2016 quadrennium. Available from: http://doi.org/10.18605/2175-7275/cereus.v13n2p84-98. Access in: Dec. 2021.

- Miranda, E. F.: ORCID 0000-0003-1200-794X
  Stephany, S.: ORCID 0000-0002-6302-4259

APPENDIX B - IMPLEMENTATIONS

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec odio elit, dictum in, hendrerit sit amet, egestas sed, leo. Praesent feugiat sapien aliquet odio. Integer vitae justo. Aliquam vestibulum fringilla lorem. Sed neque lectus, consectetuer at, consectetuer sed, eleifend ac, lectus. Nulla facilisi. Pellentesque eget lectus. Proin eu metus. Sed porttitor. In hac habitasse platea dictumst. Suspendisse eu lectus. Ut mi mi, lacinia sit amet, placerat et, mollis vitae, dui. Sed ante tellus, tristique ut, iaculis eu, malesuada ac, dui. Mauris nibh leo, facilisis non, adipiscing quis, ultrices a, dui.

Morbi luctus, wisi viverra faucibus pretium, nibh est placerat odio, nec commodo wisi enim eget quam. Quisque libero justo, consectetuer a, feugiat vitae, porttitor eu, libero. Suspendisse sed mauris vitae elit sollicitudin malesuada. Maecenas ultricies eros sit amet ante. Ut venenatis velit. Maecenas sed mi eget dui varius euismod. Phasellus aliquet volutpat odio. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Pellentesque sit amet pede ac sem eleifend consectetuer. Nullam elementum, urna vel imperdiet sodales, elit ipsum pharetra ligula, ac pretium ante justo a nulla. Curabitur tristique arcu eu metus. Vestibulum lectus. Proin mauris. Proin eu nunc eu urna hendrerit faucibus. Aliquam auctor, pede consequat laoreet varius, eros tellus scelerisque quam, pellentesque hendrerit ipsum dolor sed augue. Nulla nec lacus.

Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetuer odio sem sed wisi.

Sed feugiat. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Ut pellentesque augue sed urna. Vestibulum diam eros, fringilla et, consectetuer eu, nonummy id, sapien. Nullam at lectus. In sagittis ultrices mauris. Curabitur malesuada erat sit amet massa. Fusce blandit. Aliquam erat volutpat. Aliquam euismod. Aenean vel lectus. Nunc imperdiet justo nec dolor.

Etiam euismod. Fusce facilisis lacinia dui. Suspendisse potenti. In mi erat, cursus id, nonummy sed, ullamcorper eget, sapien. Praesent pretium, magna in eleifend egestas, pede pede pretium lorem, quis consectetuer tortor sapien facilisis magna. Mauris quis magna varius nulla scelerisque imperdiet. Aliquam non quam. Aliquam porttitor quam a lacus. Praesent vel arcu ut tortor cursus volutpat. In vitae pede quis diam bibendum placerat. Fusce elementum convallis neque. Sed dolor orci, scelerisque ac, dapibus nec, ultricies ut, mi. Duis nec dui quis leo sagittis commodo.

Aliquam lectus. Vivamus leo. Quisque ornare tellus ullamcorper nulla. Mauris porttitor pharetra tortor. Sed fringilla justo sed mauris. Mauris tellus. Sed non leo. Nullam elementum, magna in cursus sodales, augue est scelerisque sapien, venenatis congue nulla arcu et pede. Ut suscipit enim vel sapien. Donec congue. Maecenas urna mi, suscipit in, placerat ut, vestibulum ut, massa. Fusce ultrices nulla et nisl.

Etiam ac leo a risus tristique nonummy. Donec dignissim tincidunt nulla. Vestibulum rhoncus molestie odio. Sed lobortis, justo et pretium lobortis, mauris turpis condimentum augue, nec ultricies nibh arcu pretium enim. Nunc purus neque, placerat id, imperdiet sed, pellentesque nec, nisl. Vestibulum imperdiet neque non sem accumsan laoreet. In hac habitasse platea dictumst. Etiam condimentum facilisis libero. Suspendisse in elit quis nisl aliquam dapibus. Pellentesque auctor sapien. Sed egestas sapien nec lectus. Pellentesque vel dui vel neque bibendum viverra. Aliquam porttitor nisl nec pede. Proin mattis libero vel turpis. Donec rutrum mauris et libero. Proin euismod porta felis. Nam lobortis, metus quis elementum commodo, nunc lectus elementum mauris, eget vulputate ligula tellus eu neque. Vivamus eu dolor.

Nulla in ipsum. Praesent eros nulla, congue vitae, euismod ut, commodo a, wisi. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Aenean nonummy magna non leo. Sed felis erat, ullamcorper in, dictum non, ultricies ut, lectus. Proin vel arcu a odio lobortis euismod. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Proin ut est. Aliquam odio. Pellentesque massa turpis, cursus eu, euismod nec, tempor congue, nulla. Duis viverra gravida mauris. Cras tincidunt. Curabitur eros ligula, varius ut, pulvinar in, cursus faucibus, augue.

Nulla mattis luctus nulla. Duis commodo velit at leo. Aliquam vulputate magna et leo. Nam vestibulum ullamcorper leo. Vestibulum condimentum rutrum mauris. Donec id mauris. Morbi molestie justo et pede. Vivamus eget turpis sed nisl cursus tempor. Curabitur mollis sapien condimentum nunc. In wisi nisl, malesuada at, dignissim sit amet, lobortis in, odio. Aenean consequat arcu a ante. Pellentesque porta elit sit amet orci. Etiam at turpis nec elit ultricies imperdiet. Nulla facilisi. In hac habitasse platea dictumst. Suspendisse viverra aliquam risus. Nullam pede justo, molestie nonummy, scelerisque eu, facilisis vel, arcu.

Curabitur tellus magna, porttitor a, commodo a, commodo in, tortor. Donec interdum. Praesent scelerisque. Maecenas posuere sodales odio. Vivamus metus lacus, varius quis, imperdiet quis, rhoncus a, turpis. Etiam ligula arcu, elementum a, venenatis quis, sollicitudin sed, metus. Donec nunc pede, tincidunt in, venenatis vitae, faucibus vel, nibh. Pellentesque wisi. Nullam malesuada. Morbi ut tellus ut pede tincidunt porta. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam congue neque id dolor.

Donec et nisl at wisi luctus bibendum. Nam interdum tellus ac libero. Sed sem justo, laoreet vitae, fringilla at, adipiscing ut, nibh. Maecenas non sem quis tortor eleifend fermentum. Etiam id tortor ac mauris porta vulputate. Integer porta neque vitae massa. Maecenas tempus libero a libero posuere dictum. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Aenean quis mauris sed elit commodo placerat. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Vivamus rhoncus tincidunt libero. Etiam elementum pretium justo. Vivamus est. Morbi a tellus eget pede tristique commodo. Nulla nisl. Vestibulum sed nisl eu sapien cursus rutrum.

Nulla non mauris vitae wisi posuere convallis. Sed eu nulla nec eros scelerisque pharetra. Nullam varius. Etiam dignissim elementum metus. Vestibulum faucibus, metus sit amet mattis rhoncus, sapien dui laoreet odio, nec ultricies nibh augue a enim. Fusce in ligula. Quisque at magna et nulla commodo consequat. Proin accumsan imperdiet sem. Nunc porta. Donec feugiat mi at justo. Phasellus facilisis ipsum quis ante. In ac elit eget ipsum pharetra faucibus. Maecenas viverra nulla in massa.

Nulla ac nisl. Nullam urna nulla, ullamcorper in, interdum sit amet, gravida ut, risus. Aenean ac enim. In luctus. Phasellus eu quam vitae turpis viverra pellentesque. Duis feugiat felis ut enim. Phasellus pharetra, sem id porttitor sodales, magna nunc aliquet nibh, nec blandit nisl mauris at pede. Suspendisse risus risus, lobortis eget, semper at, imperdiet sit amet, quam. Quisque scelerisque dapibus nibh. Nam enim. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nunc ut metus. Ut metus justo, auctor at, ultrices eu, sagittis ut, purus. Aliquam aliquam.

Etiam pede massa, dapibus vitae, rhoncus in, placerat posuere, odio. Vestibulum luctus commodo lacus. Morbi lacus dui, tempor sed, euismod eget, condimentum at, tortor. Phasellus aliquet odio ac lacus tempor faucibus. Praesent sed sem. Praesent iaculis. Cras rhoncus tellus sed justo ullamcorper sagittis. Donec quis orci. Sed ut tortor quis tellus euismod tincidunt. Suspendisse congue nisl eu elit. Aliquam tortor diam, tempus id, tristique eget, sodales vel, nulla. Praesent tellus mi, condimentum sed, viverra at, consectetuer quis, lectus. In auctor vehicula orci. Sed pede sapien, euismod in, suscipit in, pharetra placerat, metus. Vivamus commodo dui non odio. Donec et felis.

Etiam suscipit aliquam arcu. Aliquam sit amet est ac purus bibendum congue. Sed in eros. Morbi non orci. Pellentesque mattis lacinia elit. Fusce molestie velit in ligula. Nullam et orci vitae nibh vulputate auctor. Aliquam eget purus. Nulla auctor wisi sed ipsum. Morbi porttitor tellus ac enim. Fusce ornare. Proin ipsum enim, tincidunt in, ornare venenatis, molestie a, augue. Donec vel pede in lacus sagittis porta. Sed hendrerit ipsum quis nisl. Suspendisse quis massa ac nibh pretium cursus. Sed sodales. Nam eu neque quis pede dignissim ornare. Maecenas eu purus ac urna tincidunt congue.

Donec et nisl id sapien blandit mattis. Aenean dictum odio sit amet risus. Morbi purus. Nulla a est sit amet purus venenatis iaculis. Vivamus viverra purus vel magna. Donec in justo sed odio malesuada dapibus. Nunc ultrices aliquam nunc. Vivamus facilisis pellentesque velit. Nulla nunc velit, vulputate dapibus, vulputate id, mattis ac, justo. Nam mattis elit dapibus purus. Quisque enim risus, congue non, elementum ut, mattis quis, sem. Quisque elit.

Maecenas non massa. Vestibulum pharetra nulla at lorem. Duis quis quam id lacus dapibus interdum. Nulla lorem. Donec ut ante quis dolor bibendum condimentum. Etiam egestas tortor vitae lacus. Praesent cursus. Mauris bibendum pede at elit. Morbi et felis a lectus interdum facilisis. Sed suscipit gravida turpis. Nulla at lectus. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Praesent nonummy luctus nibh. Proin turpis nunc, congue eu, egestas ut, fringilla at, tellus. In hac habitasse platea dictumst.

Vivamus eu tellus sed tellus consequat suscipit. Nam orci orci, malesuada id, gravida nec, ultricies vitae, erat. Donec risus turpis, luctus sit amet, interdum quis, porta sed, ipsum. Suspendisse condimentum, tortor at egestas posuere, neque metus tempor orci, et tincidunt urna nunc a purus. Sed facilisis blandit tellus. Nunc risus sem, suscipit nec, eleifend quis, cursus quis, libero. Curabitur et dolor. Sed vitae sem. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Maecenas ante. Duis ullamcorper enim. Donec tristique enim eu leo. Nullam molestie elit eu dolor. Nullam bibendum, turpis vitae tristique gravida, quam sapien tempor lectus, quis pretium tellus purus ac quam. Nulla facilisi.

Duis aliquet dui in est. Donec eget est. Nunc lectus odio, varius at, fermentum in, accumsan non, enim. Aliquam erat volutpat. Proin sit amet nulla ut eros consectetuer cursus. Phasellus dapibus aliquam justo. Nunc laoreet. Donec consequat placerat magna. Duis pretium tincidunt justo. Sed sollicitudin vestibulum quam. Nam quis ligula. Vivamus at metus. Etiam imperdiet imperdiet pede. Aenean turpis. Fusce augue velit, scelerisque sollicitudin, dictum vitae, tempor et, pede. Donec wisi sapien, feugiat in, fermentum ut, sollicitudin adipiscing, metus.

Donec vel nibh ut felis consectetuer laoreet. Donec pede. Sed id quam id wisi laoreet suscipit. Nulla lectus dolor, aliquam ac, fringilla eget, mollis ut, orci. In pellentesque justo in ligula. Maecenas turpis. Donec eleifend leo at felis tincidunt consequat. Aenean turpis metus, malesuada sed, condimentum sit amet, auctor a, wisi. Pellentesque sapien elit, bibendum ac, posuere et, congue eu, felis. Vestibulum mattis libero quis metus scelerisque ultrices. Sed purus.

Donec molestie, magna ut luctus ultrices, tellus arcu nonummy velit, sit amet pulvinar elit justo et mauris. In pede. Maecenas euismod elit eu erat. Aliquam augue wisi, facilisis congue, suscipit in, adipiscing et, ante. In justo. Cras lobortis neque ac ipsum. Nunc fermentum massa at ante. Donec orci tortor, egestas sit amet, ultrices eget, venenatis eget, mi. Maecenas vehicula leo semper est. Mauris vel metus. Aliquam erat volutpat. In rhoncus sapien ac tellus. Pellentesque ligula.

Cras dapibus, augue quis scelerisque ultricies, felis dolor placerat sem, id porta velit odio eu elit. Aenean interdum nibh sed wisi. Praesent sollicitudin vulputate dui. Praesent iaculis viverra augue. Quisque in libero. Aenean gravida lorem vitae sem ullamcorper cursus. Nunc adipiscing rutrum ante. Nunc ipsum massa, faucibus sit amet, viverra vel, elementum semper, orci. Cras eros sem, vulputate et, tincidunt id, ultrices eget, magna. Nulla varius ornare odio. Donec accumsan mauris sit amet augue. Sed ligula lacus, laoreet non, aliquam sit amet, iaculis tempor, lorem. Suspendisse eros. Nam porta, leo sed congue tempor, felis est ultrices eros, id mattis velit felis non metus. Curabitur vitae elit non mauris varius pretium. Aenean lacus sem, tincidunt ut, consequat quis, porta vitae, turpis. Nullam laoreet fermentum urna. Proin iaculis lectus.

Sed mattis, erat sit amet gravida malesuada, elit augue egestas diam, tempus scelerisque nunc nisl vitae libero. Sed consequat feugiat massa. Nunc porta, eros in eleifend varius, erat leo rutrum dui, non convallis lectus orci ut nibh. Sed lorem massa, nonummy quis, egestas id, condimentum at, nisl. Maecenas at nibh. Aliquam et augue at nunc pellentesque ullamcorper. Duis nisl nibh, laoreet suscipit, convallis ut, rutrum id, enim. Phasellus odio. Nulla nulla elit, molestie non, scelerisque at, vestibulum eu, nulla. Ut odio nisl, facilisis id, mollis et, scelerisque nec, enim. Aenean sem leo, pellentesque sit amet, scelerisque sit amet, vehicula pellentesque, sapien.

ANNEX A - STENCIL CODE

This appendix shows the original codes, developed in C by Torsten Hoefler (BALAJI et al., 2017), which were used as a reference to write the serial and parallel implementations of the stencil case study used in this work.

A.1 Stencil serial

Original untouched serial version. Uses two arrays to store the grid, one holds data during execution and the other holds the result of calculations. Two loops are used to iterate through the 2D matrix and update grid points. Three fixed array points are used to insert heat units into each repeating loop. A variable is used to accumulate the amount of heat inserted in each loop (note: in the implementation used in this work, the variable was moved out of the main loop. See subsection 4.1.2).

Listing A.1 - stencil.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

// row-major order
#define ind(i,j) (j)*n+i

void printarr(double *a, int n) {
// does nothing right now, should record each "frame" as image
FILE *fp = fopen("heat.svg", "w");
const int size = 5;

fprintf(fp, "<html>\n<body>\n<svg xmlns=\"http://www.w3.org/2000/svg\" version
    =\"1.1\">");

fprintf(fp, "\n<rect x=\"0\" y=\"0\" width=\"%i\" height=\"%i\" style=\"stroke-
    width:1;fill:rgb(0,0,0);stroke:rgb(0,0,0)\"/>", size*n, size*n);
for(int i=1; i<n+1; ++i)
for(int j=1; j<n+1; ++j) {
int rgb = (a[ind(i,j)] > 0) ? rgb = (int)round(255.0*a[ind(i,j)]) : 0.0;
if(rgb>255) rgb=255;
if(rgb) fprintf(fp, "\n<rect x=\"%i\" y=\"%i\" width=\"%i\" height=\"%i\" style
    =\"stroke-width:1;fill:rgb(%i,0,0);stroke:rgb(%i,0,0)\"/>", size*(i-1), size
    *(j-1), size, size, rgb, rgb);
}
fprintf(fp, "</svg>\n</body>\n</html>");

fclose(fp);
}

int main(int argc, char **argv) {

int n = atoi(argv[1]); // nxn grid
int energy = atoi(argv[2]); // energy to be injected per iteration
int niters = atoi(argv[3]); // number of iterations
double *aold = (double*)calloc(1,(n+2)*(n+2)*sizeof(double)); // 1-wide halo
    zones!
double *anew = (double*)calloc(1,(n+2)*(n+2)*sizeof(double)); // 1-wide halo-
    zones!
double *tmp;

MPI_Init(NULL, NULL);

#define nsources 3
int sources[nsources][2] = {{n/2, n/2}, {n/3, n/3}, {n*4/5, n*8/9}};

double heat=0.0; // total heat in system
double t=-MPI_Wtime();
for(int iter=0; iter<niters; ++iter) {
for(int j=1; j<n+1; ++j) {
for(int i=1; i<n+1; ++i) {
anew[ind(i,j)] = aold[ind(i,j)]/2.0 + (aold[ind(i-1,j)] + aold[ind(i+1,j)] + aold
    [ind(i,j-1)] + aold[ind(i,j+1)])/4.0/2.0;
heat += anew[ind(i,j)];
}
}
for(int i=0; i<nsources; ++i) {
anew[ind(sources[i][0],sources[i][1])] += energy; // heat source
}
tmp=anew; anew=aold; aold=tmp; // swap arrays
}
t+=MPI_Wtime();
printarr(anew, n);
printf("last heat: %f time: %f\n", heat, t);

MPI_Finalize();
}
```

A.2 Stencil parallel

Original untouched code from the parallel version. Divides the grid into parts, and each part is calculated by an MPI process. Communication between the processes is necessary, because to calculate an edge point, it is necessary to know the value of the point that is in the adjacent process.

Listing A.2 - stencil_mpi.c

```c
/*
* Copyright (c) 2012 Torsten Hoefler. All rights reserved.
*
* Author(s): Torsten Hoefler <htor@illinois.edu>
*
*/

#include "stencil_par.h"

int main(int argc, char **argv) {

MPI_Init(&argc, &argv);
int r,p;
MPI_Comm comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm, &r);
MPI_Comm_size(comm, &p);
int n, energy, niters, px, py;

if (r==0) {
// argument checking
if(argc < 6) {
if(!r) printf("usage: stencil_mpi <n> <energy> <niters> <px> <py>\n");
MPI_Finalize();
exit(1);
}

n = atoi(argv[1]); // nxn grid
energy = atoi(argv[2]); // energy to be injected per iteration
niters = atoi(argv[3]); // number of iterations
px=atoi(argv[4]); // 1st dim processes
py=atoi(argv[5]); // 2nd dim processes

if(px * py != p) MPI_Abort(comm, 1);// abort if px or py are wrong
if(n % py != 0) MPI_Abort(comm, 2); // abort px needs to divide n
if(n % px != 0) MPI_Abort(comm, 3); // abort py needs to divide n

// distribute arguments
```

```
int args[5] = {n, energy, niters, px,  py};
MPI_Bcast(args, 5, MPI_INT, 0, comm);
}
else {
int args[5];
MPI_Bcast(args, 5, MPI_INT, 0, comm);
n=args[0]; energy=args[1]; niters=args[2]; px=args[3]; py=args[4];
}

// determine my coordinates (x,y) -- r=x*a+y in the 2d processor array
int rx = r % px;
int ry = r / px;
// determine my four neighbors
int north = (ry-1)*px+rx; if(ry-1 < 0)   north = MPI_PROC_NULL;
int south = (ry+1)*px+rx; if(ry+1 >= py) south = MPI_PROC_NULL;
int west= ry*px+rx-1;     if(rx-1 < 0)   west = MPI_PROC_NULL;
int east = ry*px+rx+1;    if(rx+1 >= px) east = MPI_PROC_NULL;
// decompose the domain
int bx = n/px; // block size in x
int by = n/py; // block size in y
int offx = rx*bx; // offset in x
int offy = ry*by; // offset in y

//printf("%i (%i,%i) - w: %i, e: %i, n: %i, s: %i\n", r, ry,rx,west,east,north,
    south);

// allocate two work arrays
double *aold = (double*)calloc(1,(bx+2)*(by+2)*sizeof(double)); // 1-wide halo
    zones!
double *anew = (double*)calloc(1,(bx+2)*(by+2)*sizeof(double)); // 1-wide halo
    zones!
double *tmp;

// initialize three heat sources
#define nsources 3
int sources[nsources][2] = {{n/2,n/2}, {n/3,n/3}, {n*4/5,n*8/9}};
int locnsources=0; // number of sources in my area
int locsources[nsources][2]; // sources local to my rank
for (int i=0; i<nsources; ++i) { // determine which sources are in my patch
int locx = sources[i][0] - offx;
int locy = sources[i][1] - offy;
if(locx >= 0 && locx < bx && locy >= 0 && locy < by) {
locsources[locnsources][0] = locx+1; // offset by halo zone
locsources[locnsources][1] = locy+1; // offset by halo zone
locnsources++;
}
}

double t=-MPI_Wtime(); // take time
// allocate communication buffers
double *sbufnorth = (double*)calloc(1,bx*sizeof(double)); // send buffers
double *sbufsouth = (double*)calloc(1,bx*sizeof(double));
double *sbufeast = (double*)calloc(1,by*sizeof(double));
double *sbufwest = (double*)calloc(1,by*sizeof(double));
double *rbufnorth = (double*)calloc(1,bx*sizeof(double)); // receive buffers
double *rbufsouth = (double*)calloc(1,bx*sizeof(double));
double *rbufeast = (double*)calloc(1,by*sizeof(double));
double *rbufwest = (double*)calloc(1,by*sizeof(double));

double heat; // total heat in system
for(int iter=0; iter<niters; ++iter) {
// refresh heat sources
for(int i=0; i<locnsources; ++i) {
aold[ind(locsources[i][0],locsources[i][1])] += energy; // heat source
}

// exchange data with neighbors
MPI_Request reqs[8];
for(int i=0; i<bx; ++i) sbufnorth[i] = aold[ind(i+1,1)]; // pack loop - last
    valid region
for(int i=0; i<bx; ++i) sbufsouth[i] = aold[ind(i+1,by)]; // pack loop
for(int i=0; i<by; ++i) sbufeast[i] = aold[ind(bx,i+1)]; // pack loop
for(int i=0; i<by; ++i) sbufwest[i] = aold[ind(1,i+1)]; // pack loop
MPI_Isend(sbufnorth, bx, MPI_DOUBLE, north, 9, comm, &reqs[0]);
MPI_Isend(sbufsouth, bx, MPI_DOUBLE, south, 9, comm, &reqs[1]);
MPI_Isend(sbufeast, by, MPI_DOUBLE, east, 9, comm, &reqs[2]);
MPI_Isend(sbufwest, by, MPI_DOUBLE, west, 9, comm, &reqs[3]);
MPI_Irecv(rbufnorth, bx, MPI_DOUBLE, north, 9, comm, &reqs[4]);
MPI_Irecv(rbufsouth, bx, MPI_DOUBLE, south, 9, comm, &reqs[5]);
MPI_Irecv(rbufeast, by, MPI_DOUBLE, east, 9, comm, &reqs[6]);
MPI_Irecv(rbufwest, by, MPI_DOUBLE, west, 9, comm, &reqs[7]);
MPI_Waitall(8, reqs, MPI_STATUSES_IGNORE);
for(int i=0; i<bx; ++i) aold[ind(i+1,0)] = rbufnorth[i]; // unpack loop - into
    ghost cells
for(int i=0; i<bx; ++i) aold[ind(i+1,by+1)] = rbufsouth[i]; // unpack loop
for(int i=0; i<by; ++i) aold[ind(bx+1,i+1)] = rbufeast[i]; // unpack loop
for(int i=0; i<by; ++i) aold[ind(0,i+1)] = rbufwest[i]; // unpack loop

// update grid points
heat = 0.0;
```

```
for(int j=1; j<by+1; ++j) {
for(int i=1; i<bx+1; ++i) {
anew[ind(i,j)] = aold[ind(i,j)]/2.0 + (aold[ind(i-1,j)] + aold[ind(i+1,j)] + aold
    [ind(i,j-1)] + aold[ind(i,j+1)])/4.0/2.0;
heat += anew[ind(i,j)];
}
}

// swap arrays
tmp=anew; anew=aold; aold=tmp;

// optional - print image
if(iter == niters-1) printarr_par(iter, anew, n, px, py, rx, ry, bx, by, offx,
    offy, comm);
}
t+=MPI_Wtime();

// get final heat in the system
double rheat;
MPI_Allreduce(&heat, &rheat, 1, MPI_DOUBLE, MPI_SUM, comm);
if(!r) printf("[%i] last heat: %f time: %f\n", r, rheat, t);

MPI_Finalize();
}
```

Listing A.3 - stencil_par.h

```
/*
* stencil_par.h
*
*  Created on: Jan 4, 2012
*      Author: htor
*/

#ifndef STENCIL_PAR_H_
#define STENCIL_PAR_H_

#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

// row-major order
#define ind(i,j) (j)*(bx+2)+(i)
```

```c

void printarr_par(int iter, double* array, int size, int px, int py, int rx, int
    ry, int bx, int by, int offx, int offy, MPI_Comm comm);

#endif /* STENCIL_PAR_H_ */
```