COMMON MPI-BASED HPC APPROACHES IN PYTHON EVALUATED FOR SELECTED TEST CASES

Eduardo Furlan Miranda

Master's Dissertation of the Graduate Course in Applied Computing, guided by Dr. Stephan Stephany, approved in February 17, 2022.

INPE
São José dos Campos
2022

PUBLISHED BY:

Instituto Nacional de Pesquisas Espaciais - INPE
Coordenação de Ensino, Pesquisa e Extensão (COEPE)
Divisão de Biblioteca (DIBIB)
CEP 12.227-010
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/7348
E-mail: pubtc@inpe.br

COMMON MPI-BASED HPC APPROACHES IN PYTHON EVALUATED FOR SELECTED TEST CASES

Eduardo Furlan Miranda

Master's Dissertation of the Graduate Course in Applied Computing, guided by Dr. Stephan Stephany, approved in February 17, 2022.

INPE
São José dos Campos
2022

Cataloging in Publication Data

Miranda, Eduardo Furlan.

M672c Common MPI-Based HPC approaches in Python evaluated for selected test cases / Eduardo Furlan Miranda. – São José dos Campos : INPE, 2022.

xxiv + 197 p. ; (urlib.net/www/2022/02.14.13.19-TDI)

Dissertation (Master in Applied Computing) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2022.

Guiding : Dr. Stephan Stephany.

1. High performance computing. 2. Python programming environment. 3. Parallel computing. I.Title.

CDU 004.451.7

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÕES

INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

DEFESA FINAL DE DISSERTAÇÃO DE EDUARDO FURLAN MIRANDA
BANCA Nº 030/2022, REG 391860/2019

No dia 17 de fevereiro de 2022, as 09h, por teleconferência, o(a) aluno(a) mencionado(a) acima defendeu seu trabalho final (apresentação oral seguida de arguição) perante uma Banca Examinadora, cujos membros estão listados abaixo. O(A) aluno(a) foi APROVADO(A) pela Banca Examinadora, por unanimidade, em cumprimento ao requisito exigido para obtenção do Título de Mestre em Computação Aplicada. O trabalho precisa da incorporação das correções sugeridas pela Banca Examinadora e revisão final pelo(s) orientador(es).

Título: "COMMON MPI-BASED HPC APPROACHES IN PYTHON EVALUATED FOR SELECTED TEST CASES"

Membros da banca:

Dr. Celso Luiz Mendes - Presidente - INPE
Dr. Stephan Stephany - Orientador - INPE
Dr. Valdivino Alexandre de Santiago Junior - Membro Interno - INPE
Dr. Álvaro Luiz Fazenda - Membro Externo - Unifesp

sei! assinatura eletrônica

Documento assinado eletronicamente por **Valdivino Alexandre de Santiago Júnior**, **Tecnologista em Ciência e Tecnologia**, em 18/02/2022, às 09:58 (horário oficial de Brasília), com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

sei! assinatura eletrônica

Documento assinado eletronicamente por **Stephan Stephany**, **Pesquisador Titular**, em 18/02/2022, às 11:21 (horário oficial de Brasília), com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

sei! assinatura eletrônica

Documento assinado eletronicamente por **Celso Luiz mendes (E)**, **Usuário Externo**, em 23/02/2022, às 17:24 (horário oficial de Brasília), com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

sei! assinatura eletrônica

Documento assinado eletronicamente por **Alvaro Luiz Fazenda (E)**, **Usuário Externo**, em 02/03/2022, às 16:48 (horário oficial de Brasília), com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site http://sei.mctic.gov.br/verifica.html, informando o código verificador **9478415** e o código CRC **0A1418EC**.

Referência: Processo nº 01340.001116/2022-91 SEI nº 9478415

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to the National Institute for Space Research (INPE) and to its Applied Computing postgraduate course, which enabled my research. Also, I am grateful to my advisor, Stephan Stephany, for his guidance and encouragement, and to the LNCC (National Laboratory for Scientific Computing) for the 205341 AMPEMI 2020-I scholarship, which allowed access to the Santos Dumont supercomputer (SINAPAD node of the Brazilian HPC system).

The current work was accomplished with the support of the Coordination for the Improvement of Higher Education Personnel, Brazil (CAPES - Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Brasil), Funding Code (Código de Financiamento) 001.

ABSTRACT

A number of the most common MPI-based high-performance computing approaches available in the Python programming environment of the LNCC Santos Dumont supercomputer are compared using three selected test cases. Python includes specific libraries, development tools, implementations, documentation and optimization or parallelization resources. It provides a straightforward way to allow programs to be written with a high level of abstraction, but the parallelization features to exploit multiple cores, processors or accelerators such as GPUs are diverse and may not be easily selectable by the programmer. This work compares common approaches in Python to increase computing performance for three test cases: a 2D heat transfer problem solved by the finite difference method, a 3D fast Fourier transform applied to synthetic data, and asteroid classification using a random forest. The corresponding serial and parallel implementations in Fortran 90 were taken as references to compare the computational performance. In addition to the performance results, a discussion of the trade-off between easiness of programming and computational performance is included. This work is intended as a primer for using parallel HPC resources in Python.

Keywords: High performance computing. Python programming environment. Parallel computing.

SOLUÇÕES COMUNS BASEADAS EM MPI PARA PROCESSAMENTO DE ALTO DESEMPENHO EM PYTHON AVALIADAS EM CASOS DE TESTE SELECIONADOS

RESUMO

Algumas das abordagens de computação de alto desempenho mais comuns baseadas em MPI disponíveis no ambiente de programação Python do supercomputador LNCC Santos Dumont são comparadas usando três casos de teste selecionados. Python inclui bibliotecas específicas, ferramentas de desenvolvimento, implementações, documentação e recursos de otimização ou paralelização. Ele fornece uma maneira direta de permitir que programas sejam escritos com 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 compara abordagens comuns em Python para se obter processamento de alto desempenho desempenho utilizando três casos de teste: um problema de transmissão de calor bidimensional resolvido por diferenças finitas, uma transformada rápida de Fourier tridimensional aplicada a dados sintéticos e uma classificação de asteróides por floresta aleatória. As correspondentes implementações seriais e paralelas em Fortran 90 foram tomadas como referência para comparação de desempenho nesses casos de teste. Além dos resultados de desempenho, inclui-se uma discussão sobre o compromisso entre facilidade de programação e desempenho de processamento. Este trabalho pretende ser uma introdução para o uso de recursos de processamento de alto desempenho baseados em MPI para Python.

Palavras-chave: Processamento de alto desempenho. Ambiente de programação Python. Computação paralela.

LIST OF FIGURES

LIST OF TABLES

Page

CONTENTS

Page

1 INTRODUCTION

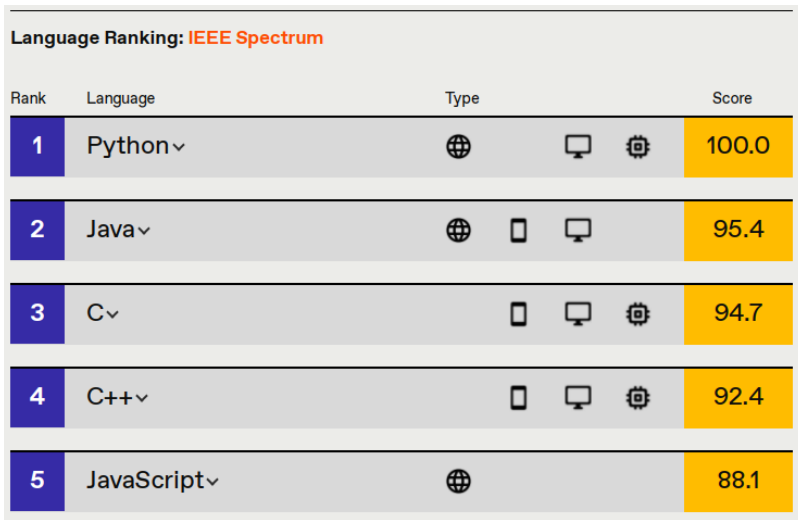
This work explores the most common high-performance computing (HPC) approaches available in the Python programming environment that are based in the Message Passing Interface (MPI) communication library. These approaches were implemented and evaluated in terms of performance for three selected test cases, which employ different algorithms and were applied to different application problems. The corresponding serial and parallel implementations in Fortran 90 (henceforth referred to as F90) were taken as references to compare the computational performance. All versions of the codes were executed in the Santos Dumont supercomputer of the LNCC (National Laboratory for Scientific Computing), henceforth referred to as SDumont. In addition to the performance results, a discussion of the trade-off between easiness of programming and computational performance is included. This work is a short primer for the use of HPC resources in the Python programming environment, using the SDumont.

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., 2020). Furthermore, Python supports 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 (PÉREZ; GRANGER, 2007), allowing for rapid prototyping. According to the 2021 IEEE Spectrum programming language ranking (IEEE Spectrum, 2021), Python is the most popular, as shown in Figure 1.1.

Python availability reaches compiler packages like the Intel one (CIELO et al., 2019) or most supercomputer programming environments. Application programs implemented in languages like F90 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). Current use of Python in supercomputing environments is exemplified in the Section 2.19 by a list of recent works.

The use of Python is also widespread for scientific applications at INPE (the Brazilian National Institute for Space Research), where its digital library lists over 80 references

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



Language Ranking: IEEE Spectrum

Rank	Language	Type	Score
1	Python	🌐 🖥 ⚙	100.0
2	Java	🌐 📱 🖥	95.4
3	C	📱 🖥 ⚙	94.7
4	C++	📱 🖥 ⚙	92.4
5	JavaScript	🌐	88.1

Source: Adapted from IEEE Spectrum (2021).

for this language (INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS (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), or the use of a neural network for the classification of supernovae (NASCIMENTO et al., 2019).

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 it difficult to analyze such trade-off in order to choose one of them.

This work aims to explore the most common MPI-based (DALCÍN et al., 2008) parallelization approaches available in the Python ecosystem, which includes libraries, frameworks and tools. The performance of these Python HPC approaches is then compared to the correspondent serial and MPI F90 implementations for three specific tests cases:

- Stencil test case: a five-point stencil finite difference method to solve partial differential equations resulting from Poisson equations, applied to a 2D heat transfer problem on a finite surface;

- Fast Fourier Transform (FFT) test case: an algorithm that computes the multidimensional Fourier transform of an 3D array of synthetic data;
- Random Forest test case: a random forest algorithm applied for the classification of asteroid orbits of a NASA dataset.

In most of this work parallelization is achieved using MPI (GROPP et al., 1996; BARNEY, 2021), but some implementations use IPython Parallel (LIMPRASERT, 2015), both for CPU execution (Central Processing Unit, which refers to a processor core). Therefore, multiple cores of the processors of one or more computing nodes are employed. Some few implementations were executed in a GPU (Graphics Processing Unit), used as an accelerator for the compute-intensive parts of a program executed in the CPU.

Some considerations about this work, as well as about Python in general, follows:

a) Python environment is very diverse, and Python code can be linked to a multitude of APIs/libraries for HPC, allowing programs to be written in many different ways;

b) Python implementations of this work include HPC solutions for standard Python (DOBESOVA, 2011), Cython (BEHNEL et al., 2010), Numba and Numba-GPU (MAROWKA, 2018a), and F2PY (PETERSON, 2009), but there are many others not employed here;

c) Python multiprocessing environment allows any parallel execution, from MPI processes to OpenMP (DAGUM; MENON, 1998) 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;

d) A current Python trend for Deep Learning is the PyTorch library (KETKAR; MOOLAYIL, 2021), which mostly generate code for execution in GPUs;

e) Standard Python code does not allow any parallelization by threads/processes (GONZALEZ et al., 2019), which is provided by Python and third-party libraries; however, in the case of thread-based libraries, there is no guarantee of thread-safeness, requiring the program avoiding race conditions, for instance using locks [1]; race conditions happen when different threads access

[1]http://www.pythontutorial.net/advanced-python/python-threading-lock

the same memory position to perform a read/write in a random order that may preclude the execution of the program in a logically correct manner;

f) Performance results shown here are specific of the selected test cases and corresponding problem size; different algorithms, applications and problem sizes may lead to a different analysis of the processing performance.

In the scope of this work, two articles were published about the Stencil test case, one in the proceedings of the XV Brazilian e-Science Workshop (BreSci-2021) (MIRANDA; STEPHANY, 2021b), and the other in the journal Cereus Magazine (MIRANDA; STEPHANY, 2021a), as shown in the Appendix A.

The remaining chapters of this document are:

- Chapter 2: Description of the MPI-based HPC approaches for Python programming employed in this work;
- Chapter 3: Description of the selected test cases, showing the corresponding implementations in Python and in F90;
- Chapter 4: Analysis of the parallel performance for the different Python and F90 implementations of the selected test cases;
- Chapter 5: Profiling for the F90 and F2PY implementations of the Stencil and FFT test cases, also estimating the overhead due to the use of Python;
- Chapter 6: Final remarks;
- Appendix A: Reference and abstract of the published articles that resulted of this work;
- Appendix B: Brief description of the Python environment;
- Appendix C: Listing and brief description of other HPC Python approaches not employed in this work;
- Appendix D: Complete set of codes implemented in this work;
- Annex A: Publicly available serial and parallel F90 codes from the Stencil test case employed in this work.

2 EMPLOYED PYTHON HPC APPROACHES

The purpose of this chapter is to briefly describe the most common MPI-based HPC approaches for Python coding employed in this work for the selected test cases. Some of these approaches are part of more general Python frameworks, like the Scikit-learn library, which is specific for machine learning, or the SciPy, for scientific computing. A complete, comprehensive description of all Python HPC resources would be not feasible in the scope of this work. Some Python HPC approaches use wrapping around existing libraries, others re-purpose existing C or F90 code, while others use compilers to generate native code. Most approaches are being continually developed and improved. A number of other Python HPC approaches, not employed in this work, are briefly described in the Appendix C.

In this work, two free and open source packages were chosen: the web application JupyterLab (Section B.6) for providing a graphical interface to the remote SDumont supercomputer, and the Conda environment and package management system (Section B.7). The following sections detail the some of the employed libraries: Scikit-learn (machine learning), SciPy (scientific computing, engineering, etc.), NumPy (mostly for array manipulation), MPI for Python and IPython Parallel (both for parallelization), Cython and Numba (both for generating optimized code), F2PY (reusing F90/C code), CuPy (execution in GPU), pyFFTW and mpi4py-fftw (serial and parallel FFT), among others. All these Python libraries/packages are free and open source.

Please observe the final sections of this chapter, addressing the SDumont computing environment for Python, including the Slurm job manager, and also describing the state-of-the-art of the use of Python in supercomputer environments.

2.1 Scikit-learn

Scikit-learn (KRAMER, 2016) is a Python library for machine learning tasks such as classification, regression or clustering via standard algorithms like support vector machine, random forest, gradient boost, k-means or DBSCAN. Scikit-learn is built on top of the SciPy library, being mainly written in Python, except for some core algorithms written in Cython to improve performance. It also uses NumPy, LIBSVM and LIBLINEAR libraries.

In addition, Scikit-learn provides easy interface with other libraries such as Matplotlib, NumPy, Pandas, SciPy, and others. Process-based or thread-based parallelism can be achieved by many different ways, according to the chosen library. For instance, in the Random Forest test case of this work, the Python version employs the Scikit-learn library with a flow of tasks supported by the joblib library, which in turn uses as parallel backend both IPP or loky. The Listing 2.1 shows an excerpt of Scikit-learn code.

Listing 2.1 - Excerpt of Scikit-learn code.

```python
from sklearn.ensemble import RandomForestClassifier
import numpy as np

# Load data
X_train = np.load("X_train.npy")
y_train = np.load("y_train.npy")

# Create a classifier
clf = RandomForestClassifier()

# Learn on the train subset
clf.fit(X_train, y_train)
```

2.2 SciPy

SciPy is a Python library used for scientific computing, mathematics, and engineering, and includes modules for optimization, linear algebra, integration, interpolation, FFT, image and signal processing, differential equation solvers, etc. SciPy has become the *de facto* standard for writing scientific computing programs in Python, with thousands of dependent packages and repositories. Other libraries, such as Scikit-learn, are built on top of it. SciPy is written using Python, Cython, Pythran, F90, C/C+, and some optimized libraries. A part of the F90 code of SciPy is a very proven legacy code, which was wrapped and transformed into a Python library. Data structures like multidimensional arrays and some other resources come from the NumPy library (VIRTANEN et al., 2020). The Listing 2.2 shows an excerpt of SciPy code.

2.3 NumPy

The NumPy library is a Python library that supports multidimensional arrays, definition of arbitrary data types, integration with databases, and a set of functions for numerical calculus, linear algebra, etc., which is not as comprehensive as the offered by SciPy. Regardless of its use in scientific computing, NumPy is more

Listing 2.2 - Excerpt of SciPy code.

```python
from scipy.fft import fft
import numpy as np

# Input array with real numbers example
x = np.array([1.7, 0.9, 0.0, -0.9, -1.7, -2.6])

# Compute the 1-D discrete Fourier transform
y = fft(x)
```

frequently used to process multidimensional data in general. NumPy has tools to integrate existing C/C++ or F90 code (WALT et al., 2011). NumPy may automatically use vectorization in order to explore processor SIMD instructions, depending on the processor.

The Python language was not conceived for numerical processing (number crunching), but its characteristics led to the development of several libraries, such as NumPy. Conversely, NumPy led to improvements in the Python syntax, such as handling arrays indexing. NumPy allow arrays to be pointed to memory addresses dynamically allocated by extensions written in C/C++ or F90, without the need to be copied, thus allowing some compatibility with existing numerical libraries, such as the linear-algebra libraries BLAS and LAPACK. The Listing 2.3 shows an excerpt of NumPy code.

Listing 2.3 - Excerpt of NumPy code.

```python
import numpy as np

# reshape gives a new shape to an array without changing its data
data = np.arange(10).reshape(2,5)
```

2.4 MPI for Python (mpi4py)

The Message Passing Interface (MPI) is the standard HPC communication library (GROPP; LUSK, 1996; UNIVERSITY OF TENNESSEE et al., 2020; DONGARRA et al., 1995). MPI for Python (mpi4py) (DALCÍN et al., 2008) is a package that provides a library with Python bindings to an MPI library that was wrapped around. In addition, mpi4py can be built choosing the underlying MPI distribution. Currently, mpi4py supports the MPI-2 standard. It makes the parallel execution of Python scripts accessible, providing most of the MPI functionality and also communication of Python objects such as NumPy arrays between processes.

Communication of Python objects not supported by the MPI standard can be done using the Python *Pickle* module, which converts the object to a sequence of bytes for the MPI communication and subsequently reconverts the sequence back to the original object (the object is *pickled* into a sequence of bytes to later be *unpickled*). The syntax of MPI for Python is similar to the MPI syntax, but does not have the MPI_ prefix, and adopting an upper-case initial letter for communication functions that support general Python objects (for example, *Send*), while using a lower case initial letter for standard MPI objects (for example, *send*). MPI for Python also supports parallel input and output in the MPI-2 standard in order to exploit parallel file systems. The Listing 2.4 shows an excerpt of mpi4py code.

Listing 2.4 - Excerpt of mpi4py code.

```python
# Write to a file test01.py and run with
# $ mpiexec -n 2 python test01.py

from mpi4py import MPI

# Communication information
comm = MPI.COMM_WORLD
mpirank = comm.Get_rank()
mpisize = comm.size

# Performs an action depending on the process or rank
if not mpirank:
    data = [1.7, 0.9, 0.0, -0.9, -1.7, -2.6]
    comm.send(data, dest=1, tag=123)
elif mpirank == 1:
    data = comm.recv(source=0, tag=123)
    print(mpisize, data)
```

2.5 IPython Parallel (IPP)

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 (IPP) (LIMPRASERT, 2015), which provides an abstraction layer that supports interactive parallel processing. IPP allows configuring a parallel execution environment for a specific architecture. Applications can be developed, executed, monitored and debugged in an interactive way. If the communication overhead is high, the programmer can employ IPP with MPI to optimize inter-process communication in addition to native IPP communication. However, in this work the standard parallel features of IPP were used.

An IPP Client object is created when there is a request to execute a parallel Python program. The request is sent to the Controller, which is composed by a Hub process and a set of Scheduler processes [1]. The Controller manages the set of Engine processes trying to meet the demand of the Client. It keeps monitoring the status of these Engines, checking their availability in order to schedule them, in a way that different Client requests may be queued and then executed. These processes are managed by Slurm. Typically, each process runs on a processor core, similarly to MPI processes. IPP also provides interactivity, since the IPP Controller is continuously monitoring new tasks and assigning them to idling IPP Engines. It provides fast, interactive parallelization with few lines of code in the case of *embarrassingly parallel* algorithms, which are trivially parallelized since there are no data dependencies.

In this work, IPP was only used in the Random Forest test case that used the corresponding algorithm of the Scikit-learn library, employing the joblib library with the IPP parallel backend. Listing 2.5 shows an excerpt of IPP code.

2.6 Cython

Cython is a compiler for the Python language, and for its own Cython extensions, which allows generating C-compiled code automatically from Python code. The C static compiler provides a more optimized code, in comparison to the original Python code (BEHNEL et al., 2010). Cython source code is compiled to the C language, which is then compiled again to generate an executable machine code. The standard operating system C-compiler is employed.

Cython can be customized by choosing specific Cython extensions for the Python language. Thus, Cython not only has interfaces for the libraries called in the original Python code, but also allows interfacing with other C/C++ codes or libraries. Cython combines the Python fast development environment with the performance of C compiled programs.

Cython compiles the original Python code providing hints about parts of the code

[1]http://tw.pycon.org/2014apac/zh/program/36.html

Listing 2.5 - Excerpt of IPP code.

```python
import ipyparallel as ipp
from ipyparallel.joblib import IPythonParallelBackend
from joblib import Parallel, parallel_backend, register_parallel_backend
from sklearn.ensemble import RandomForestClassifier
import pandas as pd, numpy as np

# Prepare the engines
c = ipp.Client(profile = "profilename")
bview = c.load_balanced_view()
register_parallel_backend('ipyparallel',
    lambda : IPythonParallelBackend(view = bview))

# Load data
X_train = np.load("X_train.npy")
y_train = np.load("y_train.npy")

# Create a random forest classifier
clf = RandomForestClassifier()

# Train the model using the training sets, in parallel
with parallel_backend('ipyparallel'):
    clf.fit(X_train, y_train)

# End
c.shutdown(hub=True, block=False)
```

that can be optimized by C-compilation, and about optimization choices. However, for these parts, it is up to the programmer to add, for example, variable type annotations to the Python code to comply with the strong typing of the C language. It is possible to add further annotations related to the optimization hints. Cython is commonly used to build Python libraries from Python code that uses Cython extensions. Therefore, the new module/library can be called from the standard Python code. The final performance will depend on the Cython compiler options, the set of extensions used, the libraries being used, or even the portability of the Python code to Cython.

The Listing 2.6 shows an excerpt of Cython code.

2.7 Numba

Numba (MAROWKA, 2018a) is usually employed as a JIT (just-in-time compiler) that converts a subset of Python and NumPy library functions into optimized machine code using the LLVM compiler infrastructure project (LATTNER; ADVE, 2004; LAM et al., Listing 2.6 - Excerpt of Cython code.

```
%%cython --force --compile-args=-O3
#cython: language_level=3

# This example uses cythonmagic, a IPython magic command interface for
# interactive work with Cython, and %%cython to compile and import a
# JupyterLab notebook cell with Cython code
import numpy as np

a = np.zeros((8, 8), np.double, 'F')
for _ in range(8):
    a += 1
print(a)
```

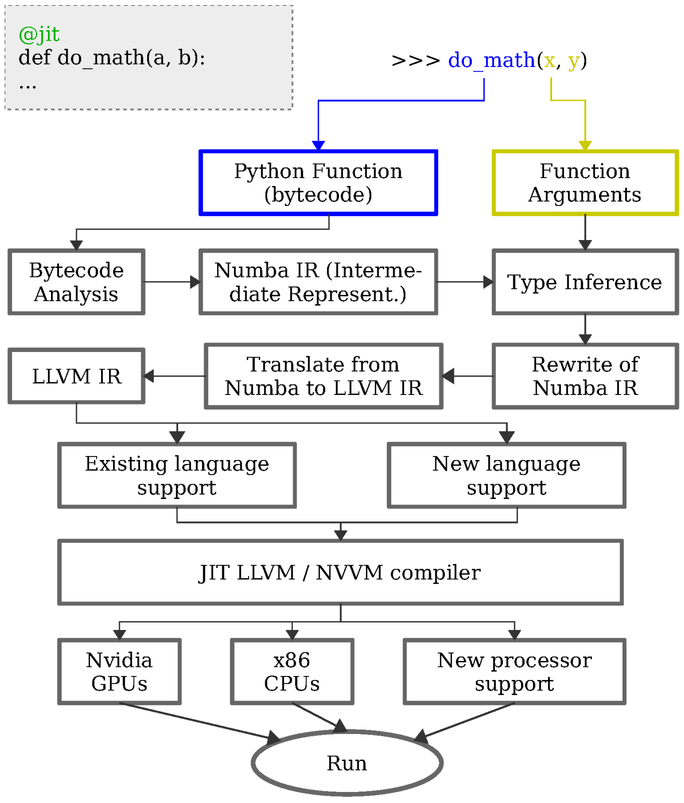
2015). LLVM is a collection of modular, reusable compiler and toolchain technologies, which began development in 2000 at the University of Illinois at Urbana-Champaign, and which can be used, as in Numba, to translate into machine code, to run on CPU or GPU. Figure 2.1 shows the diagram representing the phases of interpretation and JIT compilation of Numba.

Numba is available in the Python Anaconda distribution, and allows optimized code generation, with generally only minor changes to the original Python code. 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).

Most HPC approaches for Python employ of AOT (ahead-of-time) compilers, i.e., using code that was compiled before execution, but besides AOT, Numba also supports JIT (just-in-time) compilation during the program execution. One of the advantages of JIT compilation is portability to a different machine, with the Numba compiler producing code optimized for the specific architecture. One of the reasons for using Numba with AOT compilation is to use it on machines that may not have the Numba compiler installed.

It is important to stress the different procedure for using Numba with JIT or AOT compilation. JIT is the preferred form as it allows for portable code that employs machine-optimized Numba compilation. In the case of JIT, specific decorators must be included in the original compute-intensive Python functions in order to signal the

Figure 2.1 - Diagram of Numba JIT interpretation and compilation phases.



Source: Adapted from Lam and Seibert (2019).

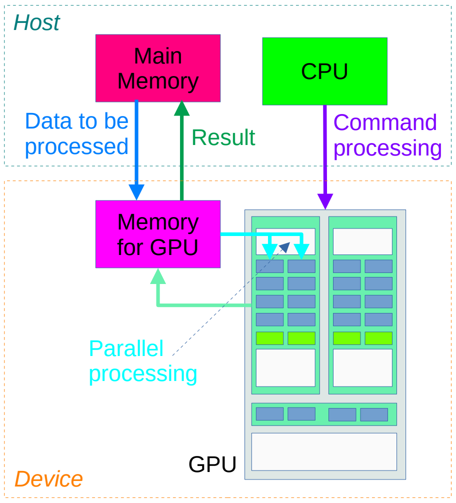
Numba compiler in execution time. In the case of AOT, it is adopted the standard approach of compiling these functions and wrapping them into a standard Python library. Numba also allows execution using a GPU, since it supports part of the Nvidia CUDA API, requiring as usual the definition of a kernel function that to be executed in the GPU, but using Python language, instead of using the CUDA extensions.

2.7.1 GPU in short

In this work, just a few test cases were executed using GPU, and thus this section contains a short introduction to such accelerators.

As shown in Figure 2.2, the GPU is the device (or processing accelerator) composed of hundreds of cores and having its own memory. As any accelerator, the GPU is part of the computing node, called the host. Typically, a node has two multicore processors (CPUs), and one or more GPUs. The kernel function contains the instructions to be executed in the GPU cores. Input data (operands) must be copied from the main

Figure 2.2 - GPU processing flow.



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

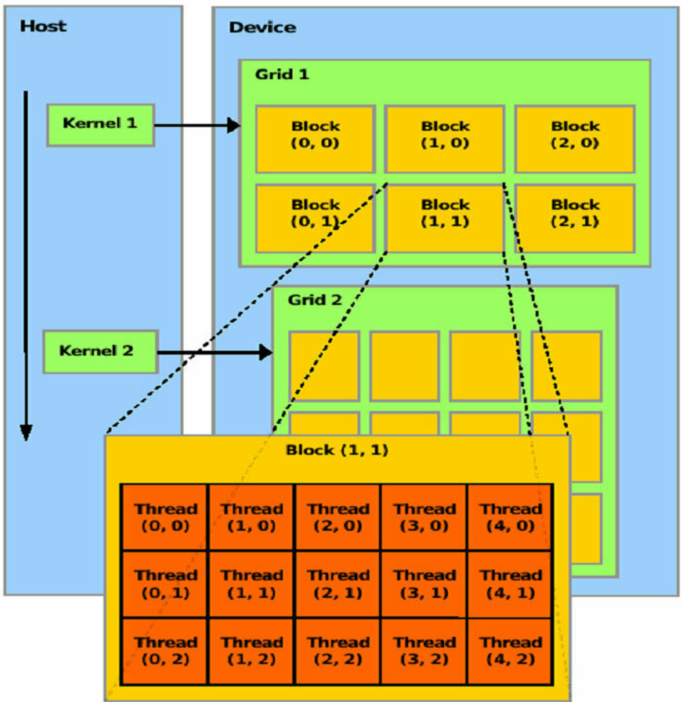
memory to the GPU memory, and after execution, output data (results) must be copied from the GPU memory to the main memory. These copies in both directions imply in significant overheads that penalize GPU performance. There are schemes to minimize such overhead, but are out of the scope of this work. The architecture of the GPU is composed by a set of streaming multiprocessors (SM), each one composed of the same number of cores. There is a global GPU memory, but also each SM have its own memory, and there are levels of cache between the global and the SM memories.

The GPU parallelization of the kernel function is achieved by mapping the problem domain into blocks of threads. The blocks are then divided into warps of usually 32 threads. Warps of the same block are assigned to one of the streaming multiprocessors of the GPU (Figure 2.3). The single-instruction multiple-threads (SIMT) paradigm models the GPU execution, since threads of the same warp are executed simultaneously. Optimized GPU execution requires dividing the domain into blocks according to the GPU architecture, i.e., taking into account the number of SMs, and to minimize memory traffic between host and device.

2.8 F2PY

F2PY (F90 for Python) allows wrapping existing optimized F90/C compiled code into a Python library (PETERSON, 2009). Thus, it allows reuse of F90/C optimized code. However, if such code is not available, the original compute-intensive part of the Python code can be rewritten in F90/C, and wrapped into a Python library by F2PY. F2PY is part of the NumPy library. The Listing 2.7 shows an excerpt of

Figure 2.3 - Execution illustrating blocks of 15 threads.



Source: Adapted from Daniel and Mircea (2010).

F2PY code.

Listing 2.7 - Excerpt of F2PY code.

```fortran
%%fortran
! This example uses fortranmagic in a JupyterLab notebook cell, which
! compiles and imports symbols from a cell with Fortran code, using F2PY.
subroutine example(a, b, c)
    real, intent(in)  :: a, b
    real, intent(out) :: c
    c = a + b
end subroutine example
```

2.9 Pandas

Pandas (MCKINNEY et al., 2011) is a package for working with relational or labeled data for data analysis and manipulation, featuring optimized manipulation of numerical tables, spreadsheets, relational databases and time series. Pandas is based on the use of DataFrame objects, providing a high level of abstraction for reading, manipulating, aggregating, and displaying data. Pandas includes statistical and other data functions, allows the importing/exporting of data from/to different file formats (CSV, QSL, Microsoft Excel, and others), and handling of missing data, filtering, reshaping, rotating, or indexing data, besides handling time series data. Pandas is performance-optimized as it includes compute-intensive parts written in Cython, and is built on top of NumPy. The Listing 2.8 shows an excerpt of Pandas code.

Listing 2.8 - Excerpt of Pandas code.

```python
import pandas as pd

# Create and display a DataFrame
df = pd.DataFrame({'Name' : ['Robert', 'John', 'Michael'],
                   'Rank' : [2, 3, 4]})
display(df)
```

2.10 pyFFTW

pyFFTW (GOMERSALL, 2021) is a Python library, which is a wrapper for the standard C-language FFTW – Fast Fourier Transform in the West (FFTW) (FRIGO; JOHNSON, 1998), a library developed at the Massachusetts Institute of Technology (MIT). The pyFFTW library performs a planning and configuration step before calculating the FFT, in order to optimize the processing performance. Consequently, pyFFT is more efficient than the simpler NumPy FFT standard module, for instance. The Listing 2.9 shows an excerpt of pyFFTW code.

Listing 2.9 - Excerpt of pyFFTW code.

```python
import numpy as np, pyfftw as pf

# Create data
data = [1.7, 0.9, 0.0, -0.9, -1.7, -2.6]

# FFT transform
result = pf.interfaces.numpy_fft.fftn(data)

# Show the result
print(result)
```

2.11 mpi4py-fft

Mpi4py-fft (MORTENSEN et al., 2019), like pyFFTW, is a Python library for calculating Fast Fourier Transforms (FFTs), but it allows parallelization through MPI to Python (mpi4py), and the use of large multidimensional arrays. Similarly to pyFFTW, it is also a wrapper for the standard C-language FFTW, developed at MIT. In the case of parallelization, it allows choosing an algorithm that will be used for decomposing the domain of the multidimensional array, for example dividing the data into slabs with convenient dimensions to be assigned to the MPI processes. Mpi4py-fft requires an installed and configured MPI library. Conda, an environment and package management system, can be used to install the required mpi4py-fft dependencies. The Listing 2.10 shows an excerpt of mpi4py-fft code.

Listing 2.10 - Excerpt of mpi4py-fft code.

```python
from mpi4py_fft import PFFT, newDistArray
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

f = PFFT(comm, [8, 8, 8], dtype=np.complex128, backend='pyfftw')
u = newDistArray(f, False)
u[:,:,:] = np.random.randn(*u.shape)

# FFT
result = f.forward(u, normalize=False)
```

2.12 Joblib

The Scikit-learn library includes joblib (FAOUZI; JANATI, 2020) among other options for parallelism. Joblib is a toolset for providing lightweight Python pipeline, aiming for simple parallelism and on-demand recalculation in the sense of storing to disk and reusing previous results, especially for large NumPy arrays. The user may choose a process-based or thread-based parallel backend, such as loky, Dask, or IPP. Joblib is based on a pipeline scheme that includes stages for tasks like I/O from/to the hard disk of operands and results, or mathematical operations. Such scheme allows a concurrent execution of different tasks in different chunks of an array, for instance. Therefore, loops through a large array are quickly executed, provided that the iterations are independent. In this work, it was used only in the Random Forest test case. The Listing 2.5 in the Section 2.5 also shows an excerpt of the joblib code.

2.13 Loky

Loky (KOLESNIKOV et al., 2020) is a high-level process-based parallel library that is the default parallel backend for the joblib library of Scikit-learn, providing ease of use. Loky creates and manages a pool of worker processes to execute tasks in parallel. All processes are started using fork+exec on POSIX systems, limiting execution to a single computing node. In this work, it was used only in the Random Forest test case. The Listing 2.11 shows an excerpt of loky code.

Listing 2.11 - Excerpt of loky code.

```python
from sklearn.ensemble import RandomForestClassifier
import numpy as np

# loky maximum number of concurrently running jobs
num_cores = 6

# Load data
X_train = np.load("X_train.npy")
y_train = np.load("y_train.npy")

# Create a classifier
# By default, loky is used
clf = RandomForestClassifier(n_jobs=num_cores)

# Learn on the train subset
clf.fit(X_train, y_train)
```

2.14 CuPy

CuPy (NISHINO; LOOMIS, 2017) is a NumPy/SciPy compatible library based on the CUDA toolkit to allow execution on GPUs. It is based on other libraries also developed for GPU execution, such as cuBLAS, cuDNN, cuRand, cuSolver, cuSPARSE, cuFFT and NCCL. CuPy API has the same API as Numpy/SciPy, and allows replacing standard code of these libraries by GPU-optimized code, thus providing a similar functionality. The Listing 2.12 shows an excerpt of CuPy code.

Listing 2.12 - Excerpt of CuPy code.

```python
import cupy as cp

a = cp.arange(10).reshape(2, 5).astype('d')
b = a.sum(axis=1)
print(b)
```

2.15 PARF

PARF (BOULESTEIX et al., 2012) is an F90 library for Random Forest classification developed by Goran Topić and Tomislav Šmuc, at the Informatics and Computing Center of the Ruđer Bošković Institute, Croatia. PARF is based on the algorithm developed by Leo Breiman (University of California, Berkeley) and Adele Cutler (Utah State University). The PARF library includes routines for data handling, Random Forest configuration, training analysis and data visualization. Input data must be done in the ARFF format (Attribute-Relation File Format) of the University of Waikato, New Zealand, an ASCII text format to describe instances and attributes of each database record. PARF is now obsolete, being superseded by new libraries like the Scikit-learn library. The PARF library source codes were written for the Intel F90 compiler, and used for the serial or MPI parallel versions.

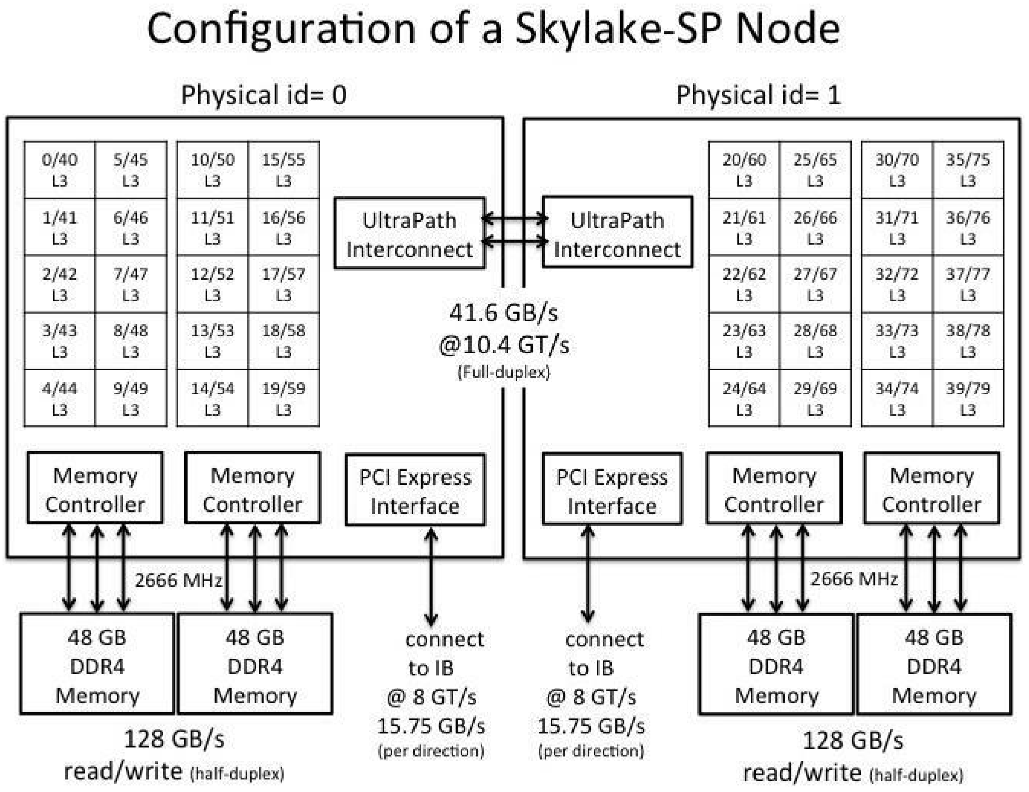
In the Random Forest test case of this work, the PARF library was directly called from the F90 amd F2PY serial and parallel implementations. In the case of the standard Python implementations, Cython and Numba implementations, the Scikit-learn library was used instead.

2.16 Optimization for NUMA

Similarly to current supercomputer shared-memory nodes, SDumont twin-processor nodes have the memory configured for NUMA (Non-UNiform Memory Access). Each processor has its local low-latency memory, composing a NUMA node, and there is an interconnection between processors to allow one processor to access the memory of the other, but with higher latency. Figure 2.4 shows an example of a NUMA memory architecture for a processing node with two Intel Skylake processors [2], similar to the ones in the SDumont Seq-X nodes (Subsection 4.1.1). NUMA optimization is intended to balance memory usage across processors to optimize memory access by assigning an equal number of threads/processes for the cores of each processor

[2]http://www.nas.nasa.gov/hecc/support/kb/skylake-processors_550.html

Figure 2.4 - Example of a processing node with two Intel Skylake processors with NUMA memory architecture (SP means Scalable Processor products of Intel).



Source: Nasa (2021).

(or close to equal in the case of an odd number). An even 50%-50% distribution of processes among the processors is advisable, in order to avoid using all cores of one processor while using only a few of the remaining processor. Such unbalance may compromise memory access in the first processor, overloading its local memory. In order to avoid this issue, a specific flag may be required in the execution of the parallel program in the command line or in the job scheduler, if the default distribution does not provide such balance.

For instance, parallel executions performed with 16 processes in 24-core B710 or 48-core Seq-X nodes (Subsection 4.1.1), such number of processes may be unevenly distributed between the processors, for instance, as [12+4] in the B710 node, and [16+0] in the Seq-X node. In order to avoid unbalancing, the Slurm script must include the *cpu_bind* option with the attribute *distribution=block:cyclic* as shown in the Listing 2.13.

Listing 2.13 - Excerpt of Slurm script.

```
> srun  −n 16 ....  −−cpu_bind=cores  −−distribution=block:cyclic ...  [executable program]
```

2.17 Python resources available on the SDumont

The SDumont computing environment provides two Python distributions, Anaconda and Intel, in addition to its default standard Python 2.7.5 version. In general, Anaconda is the most popular distribution, since it is free and open-source, including a multitude of over 7,500 packages for machine learning, data science, etc. It is also possible to install third-party packages through the use of the Conda environment and package manager. The following Anaconda distributions are available in the SDumont: anaconda2/2018.12 (Python 2.7.15), anaconda2/2019.10 (Python 2.7.16), anaconda3/2018.12 (Python 3.7.3), and anaconda3/2020.11 (Python 3.8.5). It is needed to load the corresponding operating system module.

The Intel Python distribution (CIELO et al., 2019) is a set of Python packages and libraries optimized for Intel processor architectures for scientific computing and data science applications. These optimizations are achieved by vectorization, multithreading and the use of Intel libraries designed to optimize packages like NumPy, SciPy and Scikit-learn. Intel Python includes compilers such as Numba and Cython, and libraries such as the Intel Math Kernel, Intel MPI, Intel Tread Building Blocks, and Intel Data Analytics Acceleration Library. The following Intel Python distributions are available in the SDumont: Intel Parallel Studio XE (PSXE) 2016, 2017, 2018, 2019 (Python 3.6.8), and Python 3.7.7 in Intel PSXE 2020. It is also needed to load the chosen Intel PSXE operating system module, which may require to use an Intel batch file to configure the environment.

On both Anaconda and Intel Python distributions, in the case of a missing package, Conda allows using its stacking feature or nested activation to append such package without requiring to reinstall the full Python distribution. When selecting a particular Python distribution, it is important to check its compatibility with existing libraries and/or tools. For instance, some profiling metrics of the Intel profiler are not available when using the Anaconda distribution. Conda tools allow logging the list of employed packages and versions, in order to ensure portability to other Python environments.

2.18 Using Python with the Slurm job scheduler

The Simple Linux Utility for Resource Management (Slurm) is a job scheduler used in SDumont and of common use in supercomputers and computer clusters (it may also be employed for cluster management). It is free and open source, being developed collaboratively by the Lawrence Livermore National Laboratory, and companies such as SchedMD, Linux NetworX, Hewlett Packard, and Groupe Bull, besides a large group of collaborators. Slurm allows to: (i) allocate resources such as computing nodes to users; (ii) start, execute, and monitor parallel jobs such as an MPI program on a set of allocated nodes; (iii) solve resource contention problems by managing a queue of pending jobs. It uses algorithms to optimize job allocation on the available computing nodes.

In SDumont, parallel tasks are scheduled for execution using Slurm, by means of a script file that contains all settings, options, modules, paths, etc. required by Slurm to run the executable on the computing nodes. In the case of Python, the parallel implementations of this work employ Slurm with a configuration file specific for parallel execution using MPI or IPP (Section 2.5).

In the case of MPI, each MPI process is an instance of the Python interpreter, reading the Python source code from the storage device in execution time. The computing environment must be configured using Conda prior to the parallel execution, and when the Python code ends, the MPI processes automatically terminate. In the case of IPP, the processes need to be explicitly terminated.

2.19 Current use of Python in supercomputing environments

This work has a similarity to the tutorial *Python in HPC* (RESCH, 2020) provided by the High Performance Computing (HPC) Group of the US National Institute of Health (NIH). The tutorial is aimed at those who are starting to use Python in an HPC environment, describing an example with pyOpenCL (a wrapper around OpenCL which is a framework for writing code for heterogeneous platforms) for GPU execution. There is also a discussion about disk access using Python, since it deals with small file read/write operations. Python fast code development as it is an interpreted and interactive language is emphasized, as well as the need of profiling the code to find performance bottlenecks before exploiting Python HPC resources.

Another tutorial was presented at the Exascale Computing Project 2nd Annual Meeting (2018), *Python for HPC* (SCULLIN et al., 2018). This tutorial aims to support the use of Python in some US governmental supercomputing facilities. The tutorial summarizes HPC approaches for Python, some of them employed in this work, as well as the stressing the convenience of using Python for both prototyping and implementing production software, and of using F90 to optimize high-performance kernels. There is also a discussion about the growth of the use of Python in science and technology projects, as Python is widely available in US HPC centers. Basic guidelines for HPC Python are also given to avoid excessive disk usage, and to perform code profiling or even applying the Roofline model to check if a given code is memory-bound or compute-bound for execution in the considered supercomputer.

Besides these tutorials, some articles in recent years emphasize the use of Python programming with HPC resources, as follows.

- *Towards Green Aviation with Python at Petascale* (VINCENT et al., 2016) shows the optimization of aircraft aerodynamics using Computational Fluid Dynamics (CFD) by means of the open source framework Python PyFR (WITHERDEN et al., 2014). PyFR is portable and compatible with many architectures, including AMD and Nvidia CPUs and GPUs. It uses execution time code generation to port compute-intensive kernels (parts of code that demand 50% to 85% of the processing time) from the Python intermediate language to languages such as CUDA, OpenCL, ROCm, or OpenMP/C, according to the available architecture that may combine CPUs and accelerators. Kernel specification is done by the Python Mako template engine library. It's approximately 8,000 lines of code are mainly written in the Python language. It is scalable from a laptop to a supercomputer by means of the MPI communication library. The article cites the use of Python as rapid application development of non-critical parts of code, while the overhead to execute compute-intensive kernels is minimal, generally due to the call time of an external function. The article also highlights an issue discussed in this work, about the trade-off between exploring the GPU processing power and writing code for the GPU. Due to the processing power of some GPUs, porting compute-intensive pieces of code also to CPU-executable kernel functions (hybrid processing) may require additional coding effort and complexity that may not be worth the gain in processing performance. In this way, running the compute-intensive part on the GPU can end up leaving many processor cores (CPUs) idle. There are some libraries intended to provide easy programming for GPU, like

OpenACC [3], but usually do not provide the same performance as the CUDA language.

- *Performance Analysis of Parallel Python Applications* (WAGNER et al., 2017) is about a new Python profiler, the Extrae performance monitor, which provides event-based tracing. It can be applied to Python codes with parallel backends that are thread-based (OpenMP or pthreads codes), process-based (MPI codes) or hybrid (MPI+OpenMP). It aims at obtaining profiling data as comprehensive as such provided by standard C/F90 profilers. Extrae was evaluated for an electronic structure simulation Python package used in materials science.

- *Performance evaluation of Python parallel programming models: Charm4Py and mpi4py* (FINK et al., 2021) compares mpi4py, already described in this chapter, and Charm4Py, a similar model that is based on the Charm++ object-oriented framework, which creates virtual processes to be assigned to MPI ranks. The comparison employs a set of benchmarks that include a 2D stencil problem, similar to the first test case of this work, and was executed using both CPUs and GPUs in two supercomputers, Summit and Stampede2 (respectively, #2 and #44 of the Top500 list of November 2021). Parallel scalability, granularity and load balance aspects of the tests are discussed.

- *Productivity, Portability, Performance: Data-Centric Python* (ZIOGAS et al., 2021) proposed and tested a three-layer architecture for HPC Python, composed of data-centric Python, data-centric intermediate language that provides automatic optimizations, and the processing hardware, that may include accelerators such as GPUs or FPGAs. It includes HPC extensions over annotated Python code, and thus an original Python code must be rewritten to add such annotations, but maintaining portability. The proposed approach was tested on the Piz Daint supercomputer (#20 of the Top500 list of November 2021) using a set of benchmarks and different problem sizes, showing its better parallel efficiency and scalability, when compared to other approaches such as Dask.

- *Python and HPC for High Energy Physics Data Analyzes* (SEHRISH et al., 2017) shows a test case using data from high energy physics experiments, which can easily reach up to 10 petabyte. Data is generated by a detector

[3]`http://www.openacc.org`

of subatomic particles of the Fermi National Accelerator Laboratory in dark matter research. Tabular data is provided in the HDF5 format and read into Pandas DataFrames. MPI for Python (mpi4py) is employed for parallelization, and code was executed in multicore processor nodes with/without Intel Phi accelerators.

- *GPU Computing with Python: Performance, Energy Efficiency and Usability* (HOLM et al., 2020) shows performance tests using codes and libraries for processing accelerators: CUDA for GPUs and OpenCL for GPUs and others (FPGA, DSP, etc.). The codes were written in C++ with the CUDA or OpenCL libraries, or in Python with the PyCUDA or PyOpenCL libraries. It was intended to make comparisons between the CUDA and OpenCL versions, and also between their corresponding Python versions using PyCUDA and PyOpenCL. Additional comparisons were performed for different GPUs. Some test cases have shown that the overhead of using Python is negligible, for instance comparing a PyCUDA to a CUDA version.

- *Constructing a Supercomputing Framework using Python for Hybrid Parallelism and GPU Cluster* (CHEN; YU, 2011) is about a new HPC Python software framework, called SOLVCON, for solving linear and nonlinear hyperbolic partial differential equations for Computational Fluid Dynamics (CFD) applications. Hybrid parallelism refers to execution using both CPUs and GPUs. SOLVCON also provides support for parallel I/O and visualization of the numerical results. It is organized in 5 layers that include a total of 27 modules. In typical CFD applications executed by SOLVCON, 99% of the execution time corresponds to spatial loops, which are implemented using C and/or CUDA languages. Specific SOLVCON modules provide (MPI) process-based or thread-based parallelization. The proposed approach allows writing a Python code integrating codes of other languages, libraries and tools, while obtaining a good parallel scalability.

3 SELECTED TEST CASES AND IMPLEMENTATIONS

In this chapter, each one of the test cases is defined by an algorithm and a specific application, followed by the corresponding implementations in F90 and the chosen Python HPC implementations. The three selected test cases are:

- Section 3.1: Stencil test case, applied to a heat transfer problem on a finite 2D surface solved by a finite-difference method.
- Section 3.2: FFT test case, applied to a 3D array of synthetic data.
- Section 3.3: The Random Forest test case, applied to a classification problem of asteroid orbits.

Please refer to Subsection 4.1.1 for the description of the different processing nodes of the Santos Dumont supercomputer.

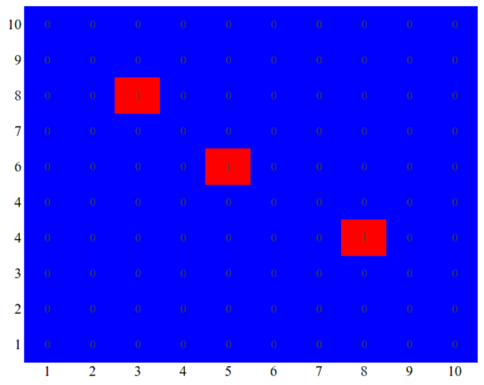
In all implementations of the different test cases, the open-source web application JupyterLab was used to experiment, develop, execute and analyze the results, including the F90 implementations. It allows the sharing of codes, data, and documents that were used to manage these implementations and allows interactive code-related functions to be implemented and executed interactively, and to check the reproducibility of the results.

3.1 Stencil test case and implementations

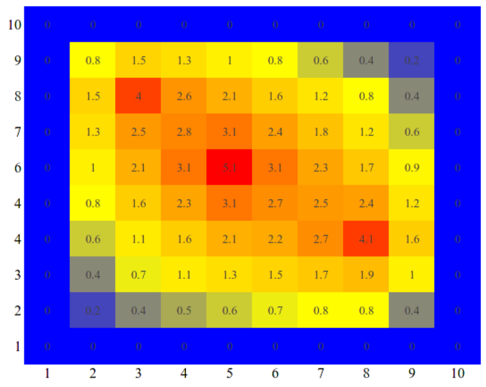
Processing performances of the Python implementations were evaluated, taking as references the serial and parallel F90 corresponding implementations. The adopted test case is a well known heat transfer problem over a finite surface (Figure 3.1), modeled by the Poisson partial-differential equation. It 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 is based on a main loop for time steps. In each iteration (time step), the 2D grid is updated and the temperature of the 3 grid points of the heat sources is increased by 1 unit, modeling the insertion of energy that is performed every time step. The updating of the 2D grid 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. A uniform temperature field with zero value is

Figure 3.1 - Initial and final temperature distribution over a finite surface exemplified for a 10×10 blue-cell grid, including constant zero-temperature boundary conditions in the outer borders, and thus the simulation encompasses only the 8×8 inner grid. Three heat sources were arbitrarily chosen, shown as **red** cells.



(a) Initial zero-temperature distribution over the 10×10 grid for a finite surface.



(b) Final temperature distribution for the same grid after 500 iterations.

Source: Author's production.

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 at localized grid points, and each introduces a unit amount of heat at each time step. The heat transfer simulation is modeled over a finite number of time steps, with all grid points being updated at each 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 allows updating a grid point by averaging the temperatures of the point itself with the temperatures of its four neighboring grid points, left-right and up-down. 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 3.1.

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} \approx \frac{U_{i+1,j} + U_{i,j+1} - 4U_{i,j} + U_{i-1,j} + U_{i,j-1}}{h^2} \tag{3.1}$$

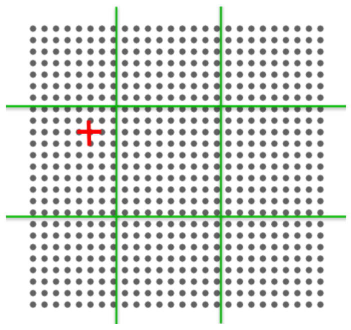
In the case of parallelization, the domain corresponding to the finite surface is divided into subdomains that are assigned to processes or threads. However, the update of points in the border of subdomains requires the temperatures values of points in the neighboring subdomains, and such data dependency between subdomains implies in communication between processes or synchronization between threads (LANGTANGEN; CAI, 2008). Considering the chosen test case, an early implementation of the algorithm was proposed by Balaji et al. (2017) using the C language, but later it was ported to F90. Figure 3.1(b) shows the final temperature distribution over a finite surface after 500 iterations, exemplified by the grid 10×10 and three arbitrarily chosen heat sources, shown as **red** cells. The simulation covers only the internal grid 8×8, and the initial zero-temperature distribution is indicated in blue.

The 2D discrete domain is shown in Figure 3.2(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 subdomains, for the sake of example. The same domain is shown in Figure 3.2(b), but with each subdomain 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 subdomain. 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 subdomain 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 subdomain from a “white point” of the subdomain above it, with the temperature of this point copied to the corresponding “yellow point” of the ghost zone of the considered central subdomain.

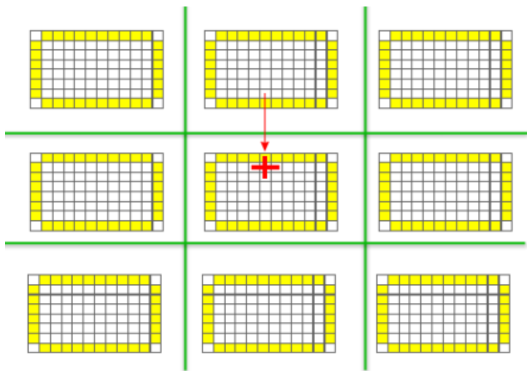
For simplification, it is assumed a square grid of $n \times n = N$ points, and also a square grid of processors or threads of $p \times p = P$ units, in a way that n is a multiple of p. It is then possible to divide the domain into P subdomains, each one containing $[N/P = (n \times n)/(p \times p) = n/p \times n/p]$ points. The ghost zone then adds two rows and two columns of grid points, considering the five-point stencil. Therefore, each subdomain contains $[(n + 2)/p \times (n + 2)/p]$ points.

Serial and parallel of the Stencil test case F90 implementations were based on a former work (SOUZA et al., 2018a). Other implementations are standard Python, Fortran-to-Python (F2PY), Cython, and Numba (including Numba-GPU). The compute-intensive part of the implementations was hand coded and does not use an existing off-the-shelf external library. In this work, most parallel versions of these

Figure 3.2 - 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).

implementations are based on MPI: the version is wrapped by the MPI for Python (mpi4py) API into a Python function, allowing the execution of MPI processes in one or more computing nodes from the Python environment. However, the F2PY API is the only exception, since it reuses a binary code generated by an F90 compiler by encapsulating it into a Python function. Parallelization using threads was restricted to the execution of a Numba JIT-compiled function in GPU, which is also discussed ahead.

Stencil test case implementations are described in the next sections.

3.1.1 F90 serial and parallel

The F90 serial and parallel versions were compiled with the GNU-compiler gfortran, which fully complies to the Fortran 95 standard. The serial F90 version is the implementation of the algorithm described in the previous section. Its corresponding parallel version employs standard MPI asynchronous non-blocking communication functions *MPI_ISend()* and *MPI_IRecv()*, which allows overlapping computation and communication, enhancing the parallel performance. However, at the end of each time step, synchronization is required for the updating of each subdomain ghost zones from the neighboring subdomains. As already mentioned, for a square grid with $N = n \times n$ points and $p \times p$ MPI processes, each process is assigned a subdomain with a total of $[(N/p) + 2] \times [(N/p) + 2]$ points, including the ghost zone.

The compute-intensive part of the code is the updating of the domain grid using the five-point stencil, as shown in the F90 code of the Listing 3.1, showing two nested loops that traverse the 2D grid, with the *anew* array stores the values of updated grid elements calculated from their previous values *aold* array.

Listing 3.1 - Compute-intensive part of the Stencil test case F90 code.

```
do j = 2, by+1
    do i = 2, bx+1
        anew(i,j) = ( aold(i, j) / 2.0 +
                    ( aold(i-1,j) + aold(i+1,j) +
                      aold(i,j-1) + aold(i,j+1) ) / 8.0 )
    enddo
enddo
```

3.1.2 F2PY serial and parallel

F2PY creates a Python library from the F90/C code, and this library is then imported by the Python code. In this test case, the function arguments of the created library are the number of grid points, the location and heat rate of the sources, the number of iterations, etc. If the F90 code is not already parallelized with MPI, a typical alternative is to use the mpi4py Python library in the Python code. F2PY seems to be convenient when an optimized F90/C code already exists and can be reused. In this test case, the Python code uses F90/C code wrapped into a library built by F2PY, as shown in the Listing 3.2. The Pyhton library libstencil is built by F2PY from F90 code, and includes the function funcstencil. The parameters *gridsize*, *energy*, *niters*, are the same as in the F90 implementation, and the total energy entered into the grid and the elapsed time are returned by the function and stored in the variables *heat* and *etime*.

Listing 3.2 - Compute-intensive part of the F2PY implementation, in the main Python code.

```
heat, etime = libstencil.funcstencil(gridsize, energy, niters)
```

3.1.3 Standard Python serial and parallel

The portability of the F90 code to Python is straightforward, requiring only NumPy as external library, which is the Python numerical-tool library. Most of the loops in the Python code can be executed using NumPy. The structure and sequence of the original code is preserved and executed interactively by the Python interpreter, while the cycle of analyzing results, changing code or parameters, and re-executing, benefits from the JupyterLab environment. However, as any interpreted language, Python is slow. Once the proof-of-concept of the Python code is complete, the code needs to be optimized, focusing on its compute-intensive parts, which are the performance bottlenecks. In this step, the programmer 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 an optimized library function. In addition, parallelization can be performed employing the native Python multiprocessing environment. In this work, Python multiprocessing was provided by the MPI for Python (mpi4py) library. The compute-intensive part of the code is the updating of the domain grid using the five-point stencil at each timestep, as shown in the Python code in the Listing 3.3. Two nested loops traverse the 2D grid, and the *anew* array stores the result of an equation that uses the elements of the *aold* array. Loops are performed internally by NumPy using the colon notation (“:”) in the array indices.

Listing 3.3 - Compute-intensive part of the Python implementation.

```
anew[1:-1,1:-1] = ( aold[1:-1,1:-1] / 2.0 +
                  ( aold[2:,1:-1] + aold[:-2,1:-1] +
                    aold[1:-1,2:]  + aold[1:-1,:-2] ) / 8.0 )
```

3.1.4 Cython serial and parallel

Cython is a compiler for both the Cython and Python languages, which is typically used to create Python libraries. These libraries are later called from standard Python code. In this test case the original Python code was reused, with few changes, in order to be compiled by Cython. The Cython parallel version employs mpi4py. The part that updates the 2D grid using the five-point stencil is shown in the Listing 3.4.

In this Cython implementation, the code included comments starting with “#cython:” that are actually compiler directives for disabling limit checking, disabling negative indexing, inferring the types of variables, among others.

Listing 3.4 - Compute-intensive part of the Cython implementation.

```
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] / 2.0 +
                        ( aold[i-1,j] + aold[i+1,j] +
                          aold[i,j-1] + aold[i,j+1] ) / 8.0 )
```

3.1.5 Numba serial and parallel

In this implementation, the compute-intensive part of the Python code was embedded into a function decorated for Numba JIT-compilation. The remaining Python code is interpreted by standard Python. Parallelization for the Numba-compiled function is provided by mpi4py for multicore processors. In the case of the standard multicore processor parallelization of the Numba code, the compute-intensive function that updates the domain grid using the five-point stencil is shown in the Listing 3.5. Loops are performed internally by NumPy using the colon notation (":") in the array indices. *@jit* is the Python decorator for Numba JIT compilation.

Listing 3.5 - Compute-intensive part of the Numba implementation.

```
@jit(nopython=True)
def kernel(anew, aold):
    anew[1:-1,1:-1] = ( aold[1:-1,1:-1] / 2.0 +
                      ( aold[2:,1:-1] + aold[:-2,1:-1] +
                        aold[1:-1,2:] + aold[1:-1,:-2] ) / 8.0 )
```

3.1.6 Numba-GPU

In this section, Numba was also employed for execution using a GPU, since Numba supports part of the Nvidia CUDA API, requiring the definition of the kernel function that will be executed on the GPU. The Numba-GPU implementation required more modifications to the standard Python serial code, than the other implementations. The compute-intensive part of the code was encapsulated into a JIT-compiled function for GPU execution by means of a Numba decorator. As usual, the remaining code of the test case algorithm is executed by the standard Python interpreter, since it is not compute-intensive.

The compute-intensive part of the code is the updating of the 2D grid/array using the five-point stencil, as shown in the code of Listing 3.6. The kernel function executed in the GPU assigns the iterations of two nested loops that traverse the 2D grid by blocks of threads in a way that each thread calculates the stencil at a grid point. Each block is divided into 32-thread warps and assigned to a particular GPU streaming multiprocessor. Two copies of the 2D array are created in the GPU memory, being swapped one for another: the *anew* array stores the 2D grid that is updated from the 2D grid stored in the *aold* array, and vice-versa along the time steps. Numba-GPU requires more code changes than the CPU version, such as defining GPU blocks and grids, and transferring data from host memory to device/GPU memory, and vice-versa. *@cuda.jit* is the Python decorator for Numba-GPU JIT compilation.

Listing 3.6 - Compute-intensive part of the Numba-GPU implementation.

```
@cuda.jit
def kernel(anew, aold):
    n = anew.shape[0] - 1
    i, j = cuda.grid(2)
    if (i > 0 and j > 0) and (i < n and j < n):
        anew[i,j] = ( aold[i,j] / 2.0 +
                    ( aold[i-1,j] + aold[i+1,j] +
                      aold[i,j-1] + aold[i,j+1]) / 8.0 )
```

The algorithm comprises the main loop with iterations that correspond to the time steps of the simulation. At each time step/iteration, the 2D array must be updated. In the serial version, such updating is entirely executed by the kernel function in the GPU. The 2D array is transferred to the GPU in the first time step, the GPU updates the 2D array and inserts energy in each time step, and only at the last time step the final 2D array is transferred back to the host memory.

However, in the parallel version, the 2D array cannot be fully updated by each MPI process (in the GPU), since the updating of the borders of its subdomain requires the updated values of the ghost zone, which are calculated by the processes that update the neighbouring subdomains. Therefore, each process needs to transfer its updated borders from the GPU to the host memory in order to send these values to the neighbouring processes. Besides, each process receives the updated borders from the neighbouring processes in order to transfer these values to the GPU, which compose the updated ghost zone of the subdomain. After the last time step is completed, the GPU of each process must transfer the final array back to its host memory.

Therefore, MPI communication and Host-GPU transfer in both directions, both related to the updating of ghost zones, make the performance of parallel versions very low compared to serial versions.

3.2 FFT test case and implementations

This section describes a specific FFT algorithm, the Fast Fourier Transform in the West (FFTW), applied to a synthetic 3D multidimensional array. An array of synthetic data has elements assigned by the programmer, as an alternative to real-world data.

FFT is an algorithm that computes the discrete Fourier transform (DFT), which is a numerical algorithm for converting a finite sequence of N equally spaced samples in the temporal or spatial domain, into the corresponding sequence in the frequency domain. For instance, the FFT allows decomposing a signal varying in time consisting of multiple pure frequencies. The Fast Fourier Transform (FFT) is an approach that reduces the DFT computation complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$.

The 1D DFT of a sequence of N complex numbers results in a sequence of $N-1$ complex numbers given by Equation 3.2.

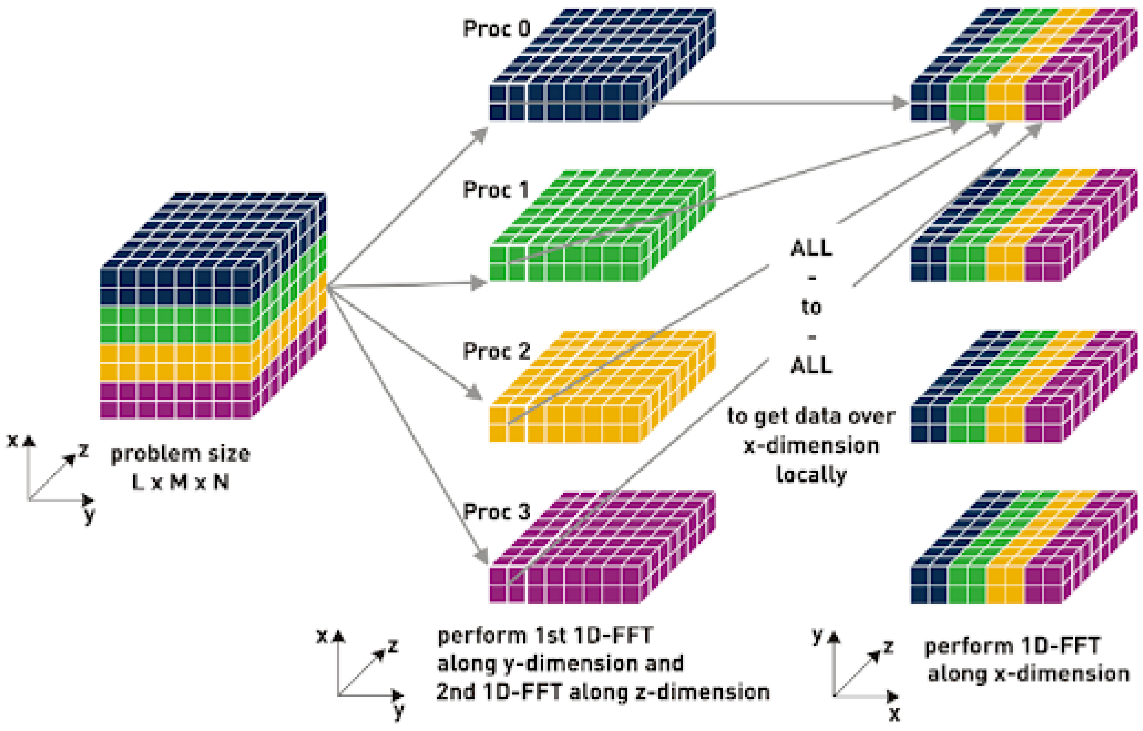
$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\dfrac{2\pi ikn}{N}} \quad , \qquad (k = 0, ..., N-1) \tag{3.2}$$

The calculation of multidimensional 3D FFTs is given by the product of the corresponding 1D FFTs along each dimension, as shown in the Equation 3.3. Each 3-element tuple of complex numbers in the time or space domain is mapped to a corresponding 3-element tuple of complex numbers in the frequency domain for the same 3D grid. The same equation can be adapted for higher dimensions.

$$X(k_1, k_2, k_3) = \sum_{n_3=0}^{N_3-1} \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x(n_1, n_2, n_3)\, e^{\frac{-2\pi ik_3n_3}{N_3} \times \frac{-2\pi ik_2n_2}{N_2} \times \frac{-2\pi ik_1n_1}{N_1}} \tag{3.3}$$

The parallelization is performed by dividing and distributing the 3D array of N^3 complex numbers into N/p slabs that are assigned to p MPI processes. Two successive 1D FFTs are performed on the y and z dimensions for each slab, and then a new set of N/p slabs is obtained by the transposition of the former ones, in a way that the new slabs are assigned to the p MPI processes. This requires an all-to-all communication.

Figure 3.3 - Parallelization of the 3D FFT, showing the decomposition of a $L \times M \times N$ domain into 4 slabs assigned to 4 MPI processes.


Proc 0
Proc 1
Proc 2
Proc 3
ALL - to - ALL
to get data over x-dimension locally
problem size L x M x N
perform 1st 1D-FFT along y-dimension and 2nd 1D-FFT along z-dimension
perform 1D-FFT along x-dimension
x
y
z

Source: Adapted from Schulz (2008).

Finally, each MPI process performs a 1D FFT on the x dimension of its slab. This scheme is shown in Figure 3.3, but considering a more generic multidimensional array of dimension $L \times M \times N$ and 4 processes,

The standard implementations of Python, Cython and Numba FFT employ the mpi4py-fft library (Section 2.11) which depends on the FFTW library, and also the mpi4py library for parallelization, and which automatically distributes large sequences or data arrays. In the case of the F90 and F2PY implementations, the FFTW library available on the SDumont was used (module *mathlibs/fftw/3.3.8_-openmpi-3.1_gnu*) with the MPI library. Therefore, differently from the Stencil test case, the compute-intensive part of the code uses off-the-shelf external libraries.

In order to obtain performance, F90 is straightforward, followed by F2PY, which required relatively few changes to the original F90 code, and Python. FFT test case implementations are described in the next sections.

3.2.1 F90 serial and parallel

The F90 serial and parallel versions were compiled with the GNU-compiler gfortran, which complies to the Fortran 95 standard. These F90 implementations employ the FFTW library available on the SDumont, and the MPI library. Parallelization is performed as described above, in the Section 3.2. The FFTW library includes a planning step before running the FFT, in order to optimize the processing performance. The compute-intensive part of the code is shown in the Listing 3.7, being performed by the FFTW library (*fftw_mpi_execute_dft*). In the code, *plan* is given as an option, *data1* is the input 3D array, while *data2* is the output one.

Listing 3.7 - compute-intensive part of the FFT test case F90 code.

```fortran
call fftw_mpi_execute_dft(plan, data1, data2)
```

3.2.2 F2PY serial and parallel

The F2PY implementation reuses the F90 source code, including the MPI and FFT libraries, with minor changes. The original compute-intensive source code is transformed into Python functions with the corresponding arguments. The F90 source code is then built using F2PY and the -O3 compilation flag, and wrapped by F2PY into a standard Python library, which contains most of the original code, including: (i) initializing the 3D array with values derived from the array indices, (ii) calculating the FFTW transform, and (iii) calculating the array checksum in order to check the correctness of the result. The remaining Python code is short, since it just imports the library to call the functions built by F2PY, and then displays the result.

Instead of passing input parameters as function arguments, another possible approach can be created by hardcoding the arguments, i.e., to declare their values as parameters inside the program. Such approach was used in this test case for convenience, since it is easy to edit the code, recompile and re-execute it using the JupyterLab notebook. Changes in the Python code are minimal, and the use of Slurm is also similar. Two different versions of the library were developed, serial and parallel, for ease of use, but is would be possible to employ a single version, and choose serial or parallel execution using the Slurm script.

The compute-intensive part of the F2PY implementation is due to the Python function calls to perform the FFTW, shown in the Listing 3.8 (Pyhton library libfft and function funcfft built by F2PY using the size *gridsize3d* of the array as argument). This function call returns the checksum of the array elements and the elapsed time, in the *csum* and *etime* variables.

Listing 3.8 - Compute-intensive part of the FFT test case Python code.

```
csum, etime = libfft.funcfft(gridsize3d)
```

3.2.3 Standard Python serial and parallel

In the Python serial implementation, the Python library pyFFTW is used, which is an encapsulated version of the C-compiled FFTW, being the compute-intensive part of the implementation. The remaining part is the standard Python code, executed in an interpreted way. However, the initialization of the 3D multidimensional array was written in Python and requires nested loops, which are very slow when executed in interpreted form. Therefore, the Python code that performs the initialization was optimized by calls to functions of the NumPy library. The parallel Python implementation simply calls the mpi4py-fft library, that adds an MPI parallelization layer to the same FFTW library.

In this implementation, the computation-intensive part of the code is shown in the Listing 3.9, being performed by the pyFFTW library (*pf.interfaces.numpy_-fft.fftn*). The array *u* contains the input 3D multidimensional array, *uf* contains the result, *overwrite_input=True* indicates that the input array can be overwritten, while *auto_contiguous=False*, and *auto_align_input=False* both indicate that the multidimensional array can be copied into contiguous memory positions that are also aligned in memory, in order to optimize memory access.

Listing 3.9 - Compute-intensive part of the FFT test case Python code.

```
uf = pf.interfaces.numpy_fft.fftn(u,
                                  overwrite_input=True,
                                  auto_contiguous=False,
                                  auto_align_input=False)
```

3.2.4 Cython serial and parallel

Cython is a static optimizing compiler that translates Python or Cython source code into C language target code that is then compiled to generate an optimized machine code. Cython implementations use the same Python libraries pyFFTW (serial), and mpi4py-fft (parallel) as the standard Python corresponding implementations. Therefore, the remaining Python code is the same for both corresponding serial and parallel implementations. Function input arguments were hardcoded, taking advantage of the JupyterLab notebook.

The compute-intensive part of the code is the function call to the library pyFFTW (serial version), or to mpi4py-fft library (parallel version), as shown in the Listing 3.10 for both versions. The *data* argument contains the input 3D multidimensional array, while *result* is the output transformed array. In the parallel version, MPI.COMM_-WORLD is the default MPI communicator, $[N, N, N]$ is the dimensions of the 3D array, dtype is the type of each element in the array (complex number), and the backend specifies the pyFFTW library, which is the same used in the serial version.

Listing 3.10 - Compute-intensive part of the FFT test case Cython code.

```
# serial version:
result = pyfftw.interfaces.numpy_fft.fftn(data)
# parallel version:
plan = PFFT(MPI.COMM_WORLD, [N, N, N], dtype=np.complex128, backend='pyfftw')
result = plan.forward(data)
```

3.2.5 Numba serial and parallel

The Numba implementation employed JIT compilation, and similarly to the standard Python and Cython implementations, the serial version uses the pyFFTW library and the parallel version uses the mpi4py-fft library. Differently from the Numba implementation of the Stencil test case, JIT compilation was not used in the computationally intensive part of the FFTW, since it is executed by the pyFFTW library which has already been AOT compiled and optimized. However, Numba was used in the rest of the code including the function that initializes the 3D multidimensional array, being faster than the corresponding part of the standard Python implementation.

The compute-intensive part of this implementation is also the function call of the library pyFFTW for the serial version, or mpi4py-fft for the parallel version, and is shown in the Listing 3.11, both already AOT compiled, for the serial and parallel versions. The parameter *data* is the array containing the input 3D multidimensional array. In the parallel version, MPI.COMM_WORLD is the standard MPI communicator, $[N, N, N]$ are the dimensions of the 3D array, dtype is the type of each element of the array (complex number), and backend specifies the pyFFTW library, which is the same used in the serial version.

Listing 3.11 - Compute-intensive part of the FFT test case Numba code.

```
# serial version:
result = pyfftw.interfaces.numpy_fft.fftn(data)
# parallel version:
plan = PFFT(MPI.COMM_WORLD, [N, N, N], dtype=np.complex128, backend='pyfftw')
result = plan.forward(data)
```

3.2.6 CuPy

The CuPy implementation was executed on a B715 node or on a Seq-X (Subsection 4.1.1), both employing a single GPU. The CuPy library is GPU-specific, NumPy-compatible, and encapsulates the CUDA toolkit. The compute-intensive part is the FFTW, performed by a CuPy library function and executed in the GPU. Other CuPy functions were employed to transfer the input array to/from the GPU memory, and to calculate the checksum of the array elements. The NumPy *fromfunction* function was used to initialize the 3D multidimensional array to avoid using the Python interpreter slow loops. The same CuPy code was executed on a B715 node using Slurm or on a Seq-X directly from the operating system command line, using a Tesla K40t GPU or a Volta V100 GPU, respectively. The compute-intensive part of the Python code of the CuPy implementation is the function call of the library routine shown in the Listing 3.12. The *data* argument is the input 3D array, and *result* is the transformed output array. Provided that there is the availability of an optimized CuPy function for the compute-intensive part of the code, this is a very simple way for employing a GPU.

Listing 3.12 - Compute-intensive part of the FFT test case CuPy code.

```
result = cupy.fft.fftn(data)
```

3.3 Random Forest test case and implementations

This test case describes the implementation of the machine learning algorithm Random Forest (RF), applied for the classification of asteroid orbits. An RF is a set of decision trees generated by an ensemble method. As in the other test cases, the corresponding RF implementations were made in F90 and Python, in both sequential and parallel versions.

3.3.1 Random Forest and ensemble methods

A decision tree is a machine learning algorithm (MLA), more specifically, a non-parametric (does not require hyperparameters) supervised (applied to known classes) learning method used for classification and regression. A decision tree is similar to a graph, composed of nodes and branches, with each node associated to an attribute of the input data, and each branch associated to the class or value of the node attribute. Nodes are ordered according to its importance for discriminating the instances. At the end of the tree, branches are assigned with the corresponding major class. A decision tree is called a regression tree if it uses numerical data, or a classification tree if it uses categorical (class) data. Similarly to other machine learning algorithms, input data is divided into information variables (input) and a decision variable (output). In the training phase, a decision tree is generated from known instances of the training dataset. Then, in the test phase, the decision tree is used to perform classification or regression on new input data in order to estimate the decision variable for each new instance of the test dataset.

An ensemble method may improve the performance of any MLA by combining a finite set of instances of the original MLA, being these instances called members of the ensemble. This ensemble-generated set of members actually represents a new MLA that is expected to yield a better result than the original MLA. In general, ensemble methods may have members being training independently one from one another *in parallel*, or being trained consecutively in sequence. Considering a given MLA, the use of an ensemble method reduces the bias error, which is due to the algorithm itself or its hyperparameters, by means of reducing the error due to the variance, which is due to sensitivity to small fluctuations in the input data. As a consequence, ensemble methods helps to avoid overfitting. The most standard ensemble methods are Bagging and Boosting, with each one having many variations.

Bagging (BREIMAN, 2001), from Bootstrap Aggregating, applies some sampling scheme to the input data in order to generate different training datasets for the ensemble members, but is possible to use an ensemble of different MLAs or of the same MLA with different hyperparameters. Each dataset contains a different set of instances, but each instance has the complete set of input attributes. Another scheme would be to use a slightly different set of attributes in the training of each member, but the complete set of instances of the input data, or yet a combination of both approaches. In addition, the use of different training data can be applied for ensembles composed of different MLAs. After the training, in the test phase, the result of the ensemble-MLA for each new instance is given by averaging the results of the members (for numerical output, in classification or regression), or by a polling scheme (for categorical output, in classification). Bagging is inherently parallelizable, since members are trained independently, by assigning one or a block of members to each MPI process or POSIX thread. An RF is an ensemble of decision trees that employs Bagging.

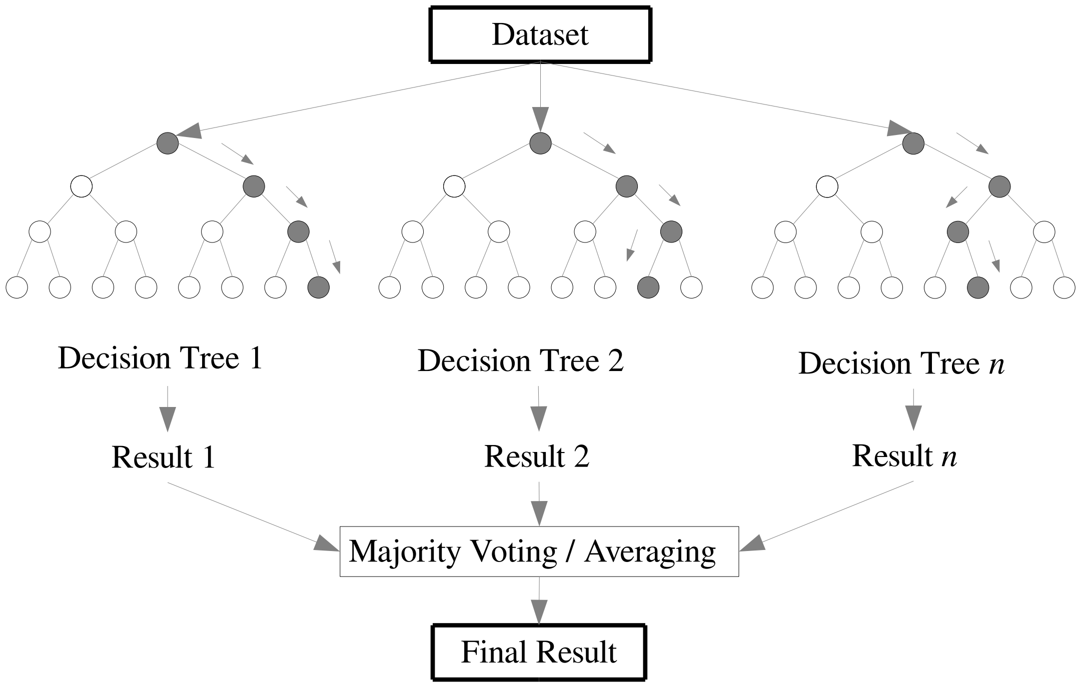
Boosting (BREIMAN, 2001) is an ensemble method in which training is performed consecutively on the members, but weighting the error of each instance. In the training of the first member, weights are initially equal to unity, but in the successive trainings of the members, the importance of each instance is weighted by its corresponding classification/regression error. Boosting it is not memberwise parallelizable, since each member is trained consecutively, but it is usually faster than Bagging, as it demands a much lower number of members. Eventually, the execution of each member may be parallelizable. After the training, in the test phase, the result of the ensemble-MLA for each new instance is given by the last member of the ensemble.

In this test case, the ensemble method is an RF, and thus bagging was applied to generate N training sets for each of the N trees. Each set is obtained by randomly sampling, with replacement, the original dataset. Figure 3.4 presents the flowchart of an RF composed of the set of decision trees in the test phase, i.e., after the decision trees were trained. Each instance from the test dataset is then classified by each of the decision trees, yielding a result, which can be numerical or categorical. The final result of the RF is then computed by averaging the numerical results or by a polling scheme for categorical results.

3.3.2 The asteroid orbit classification problem

This problem is about training an MLA, a Random Forest, to perform asteroid orbit classification. A dataset of 100,000 asteroids was randomly selected and divided into training and test sets, with respectively 66,000 and 34,000 instances, each defined in a separate ARFF file. Each instance of the dataset corresponds to an observed

Figure 3.4 - RF flowchart.



Source: Author's production.

asteroid and contains 37 attributes that the RF employs to classify the asteroid class, among the 13 possible classes for asteroid orbit, as described in Table 3.1 and in Table 3.2.

Table 3.1 - The 13 asteroid orbit classes for the *class* decision attribute in the asteroid orbit dataset.

Abbr.	**Title**	**Description**
AMO	Amor	Near-Earth asteroid orbits similar to that of 1221 Amor
APO	Apollo	Near-Earth asteroid orbits which cross the Earth's orbit
AST	Asteroid	Asteroid orbit not matching any defined orbit class
ATE	Aten	Near-Earth asteroid orbits similar to that of 2062 Aten
CEN	Centaur	Objects with orbits between Jupiter and Neptune
HYA	Hyperbolic Asteroid	Asteroids on hyperbolic orbits
IEO	Interior Earth Object	Orbit contained entirely within the orbit of the Earth
IMB	Inner Main-belt	Orbital elements constrained by Inner Main-belt
MBA	Main-belt Asteroid	Orbital elements constrained by Main-belt Asteroid
MCA	Mars-crossing Ast.	Orbit of Mars constrained by Mars-crossing
OMB	Outer Main-belt	Orbital elements constrained by Outer Main-belt
TJN	Jupiter Trojan	Traped in Jupiter's L4/L5 Lagrange points
TNO	TransNeptunian Object	Objects with orbits outside Neptune

Source: Adapted from the NASA Planetary Data System (2022).

Table 3.2 - The 37 selected information attributes plus 1 for the asteroid orbit dataset (the 38th one is the decision attribute, the class, as described in the preceding table).

Attribute	**Description**
neo	Near-Earth object
pha	Potential hazards asteroid
h	Absolute magnitude parameter
diameter	Asteroid diameter, from equivalent sphere
albedo	Geometric albedo
diameter_sigma	Diameter 1 sigma
orbit_id	Orbit id
epoch	Epoch, particular time
epoch_mjd	Epoch of the elements represented as the Modified Julian Date (MJD)
epoch_cal	Epoch calender
e	The eccentricity of the conic
a	Mean Distance, the semi-major axis of the orbit measured in AU
q	Perihelion distance [AU]
i	Inclination, the angle between the ecliptic plane and the plane of the orbit
om	Longitude of the Ascending Node (Omega)
w	Argument of Perihelion (w)
ma	Mean Anomaly (M)
ad	Aphelion distance [AU]
n	Mean motion [deg/d]
tp	Time of perihelion passage (TDB)
tp_cal	tp calender
per	Period
per_y	Period year
moid	Earth minimum orbit intersection distance au unit
moid_ld	Earth minimum orbit intersection distance lunar unit
sigma_e	e 1-sigma (see "e" above)
sigma_a	a 1-sigma
sigma_q	q 1-sigma
sigma_i	i 1-sigma
sigma_om	om 1-sigma
sigma_w	w 1-sigma
sigma_ma	ma 1-sigma
sigma_ad	ad 1-sigma
sigma_n	n 1-sigma
sigma_tp	tp 1-sigma
sigma_per	period 1-sigma
rms	A measure of the predicted data's deviation from the observed data
class	Asteroid Orbit Classes

Source: Adapted from the NASA Planetary Data System (2021).

Asteroid orbit data were obtained from the Solar System Dynamics (SSD) of the Jet Propulsion Laboratory (JPL) [1]. It provides astronomical data about the orbits, physical characteristics, and discovery circumstances for most of the known natural

[1]http://ssd.jpl.nasa.gov/

bodies in the Solar System. A subset of it, the Small Bodies Database (SBDB), provides information about small bodies like known asteroids and comets. For convenience, raw data were pre-processed using Weka software (Waikato Environment for Knowledge Analysis, developed at the University of Waikato, New Zealand) [2], and the resulting datasets are in ARFF format.

3.3.3 Random Forest implementations

In the Random Forest test case, all implementations were executed using processor cores of one or more computing nodes (no GPU). Python was used with the Scikit-learn library (Section 2.1), and since this library is a highly optimized and constantly updated, performance results were better than those obtained by the F90 or F2PY implementations, which are based on the obsolete PARF library (Section 2.15). As a consequence, differently from the previous test cases, the F90 implementation was not taken as a reference.

It is important to note that, with the exception of the F90 and F2PY implementations, the other Python implementations of this test case (standard Python, Cython and Numba) do not employ the MPI communication library for parallelization, using instead the IPP library. Parallelization is accomplished by running multiple training and prediction processes on decision trees, using multiple MPI or IPP processes (depending on implementation). In the case of Python, the Scikit-learn library uses the IPP library as a backend, which works using engines and other components that run in processes, as discussed in the Section 2.5.

Random forest test case implementations are described in the next sections.

3.3.3.1 F90 serial and parallel

The serial and MPI-based parallel F90 implementations use the PARF (Section 2.15), an F90 library written for Random Forest classification. The compute-intensive part of the code, executed by the PARF library is the *build_tree* routine that builds the trees, and corresponds to 46% of the total processing time according to the `gprof` operating system profiler.

3.3.3.2 F2PY serial and parallel

In the F2PY implementation, the original PARF F90 code with a command line interface was re-written as a subroutine and then built by F2PY into a Python

[2] `http://www.cs.waikato.ac.nz/ml/weka/`

library. The Intel compiler with the optimization flag -O3 was used instead of the GNU compiler, since PARF requires some Intel resources. The input are the files containing the datasets, and the outputs are the classification error, another metric for classification accuracy (kappa), and the elapsed time measured using the F90 library wall time function. In the parallel version, all MPI declarations and calls are in the PARF code, and the F2PY is built using the Intel MPI.

The remaining Python code is basically the same for both serial and parallel versions, being the only difference the name of the library built by F2PY for the serial and parallel versions. The number of MPI processes is defined in the Slurm script (1, 4, 16, 24, 48, 72, and 96 processes), and execution times are the average of 3 runs.

The compute-intensive part of the code shown in Listing 3.13 refers to the F2PY-built function. In the listing, *result* contains the set of output parameters, *lib_p2py_parf* is the name of the library created by P2PY, *random_forest* is the function with the compiled PARF code, and the files *train.arff* and *test.arff* contain the datasets for the training and testing phases, respectively.

Listing 3.13 - Compute-intensive part of the Random Forest test case F2PY code.

```python
result = lib_p2py_parf.random_forest("train.arff", "test.arff")
```

3.3.3.3 Standard Python serial and parallel

The standard Python serial and parallel implementations, as well as the Cython and Numba ones, use the Scikit-learn library that employs the IPP library as parallel backend. The Scikit-learn Random Forest library is faster than PARF. The Pandas library was used to store the datasets, which are read from ARFF files using the SciPy library.

This test case requires the training and test phases of a classification algorithm. The use of Scikit-learn requires a configuration step to select an estimator/classifier (in this case, the Random Forest), and also to select the specific parameters of the estimator. In the next step, training is performed using the training dataset, and then testing, using the test dataset.

The compute-intensive part of the code is shown in the Listing 3.14 and is performed by the Scikit-learn library. In the parallel version, the line that declares the backend is added. The *clf* is the Random Forest classifier, the *fit* is the function that performs the classification, and the X contains a matrix of dimension $66,000 \times 36$, since there are 66,000 training instances and 36 attributes. And finally, y is a vector of dimension 66,000 containing the known classification for these instances into one of 13 possible classes. The result is the trained Random Forest model, also stored as *clf*, which can then be employed in the test phase.

Listing 3.14 - Compute-intensive part of the Random Forest test case Python code.

```python
with parallel_backend('ipyparallel'):
    clf.fit(X, y)
```

3.3.3.4 Cython serial and parallel

In the Cython implementation, the same code of the Python standard implementation is reused, and thus includes the same calls to the Scikit-learn library. As this library is already optimized for performance, porting the Python code to Cython would not significantly improve performance. The Cython parallel version is also similar to the corresponding standard Python, using IPP as the parallel backend.

Please see the Listing 3.14 in the Subsubsection 3.3.3.3 containing the compute-intensive part of the Cython code, as it is identical to the standard Python implementation.

3.3.3.5 Numba serial and parallel

In the Numba implementation, the same code as the standard Python implementation is reused and therefore includes the same calls to the optimized Scikit-learn library. Since this library is already optimized for performance, there would be useless to port the source code to Numba in order to optimize it. However, there is a small gain of performance by using Numba JIT compilation for the remaining part of the Python code. The IPP parallel backend is also employed for the Numba parallel version. The Numba implementation is then executed in three different ways: (i) interpreted by standard Python (only for small parts of the original code), (ii) executed using optimized library functions (compute-intensive part), and (iii) executed using Numba JIT-compilation (remaining part).

Please see the Listing 3.14 in the Subsubsection 3.3.3.3 containing the compute-intensive part of the Numba code, since it is identical to the standard Python implementation.

4 TEST CASES PARALLEL PERFORMANCE

This chapter covers the analysis of the parallel performance of the different serial and parallel implementations (F90 and Python) for each one of the three selected test cases. Most of the parallel implementations are MPI-based, but there are some others that employ specific Python libraries. The only implementations that were executed using thread parallelism were Numba-GPU and CuPy for GPU execution, and the loky implementation for CPU execution using threads. This chapter is divided as follows:

- Section 4.1 Test environment: standard parallel performance definitions, description of the SDumont execution nodes, and a listing of the versions of the employed compilers, Python, etc.;
- Section 4.2 Stencil test case: a five-point stencil finite difference method to solve partial differential equations resulting from Poisson equations, applied to a 2D heat transfer problem on a finite surface;
- Section 4.3 Fast Fourier Transform (FFT) test case: an algorithm that computes the multidimensional Fourier transform of an 3D array of synthetic data;
- Section 4.4 Random Forest test case: a random forest algorithm applied for the classification of asteroid orbits of a NASA dataset.

General guidelines for the tests: all processing times shown here are the average of 3 executions. The calculation of speedups and parallel efficiencies always took the serial execution time of the F90 implementation as a reference, except for the Randon Forest test case, which used the serial execution time of the standard Python implementation as a reference. The serial and parallel test cases were executed by means of the Slurm job scheduler, except where otherwise stated. The JupyterLab interactive environment was employed in all tests.

4.1 Test environment

Standard parallel performance metrics are used here, like the speedup S_p, given by the ratio between the serial t_s and the parallel execution time t_p, using p processes or threads (Equation 4.1), being the ideal speedup, called linear speedup, equal to p.

$$S_p = \frac{t_s}{t_p} \tag{4.1}$$

In this work, for each considered implementation of any test case, the speedup is calculated using the time of the serial version and the corresponding time of the parallel version, i.e., same programming language and compiler. However, the best serial time of each test case is considered, in order to allow a more fair comparison of speedups. Another standard metric is the parallel efficiency E_p, given by the ratio between the speedup S_p and the corresponding number of p processes or threads (Equation 4.2). Thus, a linear speedup corresponds to a parallel efficiency of 100%, or unitary. It is important to note that the parallel efficiency can be higher than 100%, depending on the value adopted as a reference for the speedup calculation. It can also happen for serial and parallel versions generated with the same programming language and compiler, if the parallelization implies in an optimization of the memory access, and thus lowering execution times of each process/thread.

$$E_p = \frac{S_p}{p} \tag{4.2}$$

4.1.1 The Santos Dumont computing environment

This section describes briefly the SDumont computing environment in terms of software and hardware. Three different computer nodes [1] of the LNCC Santos Dumont supercomputer were employed:

- **Thin node B710** (B710), 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;
- **Thin node B715** (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 B710 nodes, there are CUDA libraries;
- **Sequana X node** (Seq-X)
  - **Execution node**, with 2 Intel Xeon Gold 6252 (2.1 GHz) 24-core processors (total of 48 cores per node), 4 GPUs Nvidia Volta V100 and 384 GB main memory; in addition to the compilers and libraries of the B710 nodes, there are CUDA libraries;

[1] `http://sdumont.lncc.br/machine.php?pg=machine`

– **sdumont18 login node** [2] , 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.

The employed compilers and libraries for F90 and Python are available in the SDumont computing environment. Serial and parallel implementations of the F90 used two suites of tools and compilers: GNU version 4.8.5 and Intel versions 19.0.3 or 19.1.2, included in the Intel Parallel Studio (PSXE) 2019/2020. Intel-compiled implementations used the Intel MPI library of the corresponding version, while GNU-compiled implementations used the OpenMPI library versions 3.3.8, 4.0.1 and 4.0.4. The standard optimization flag adopted in this work for F90 is -O3, which in general allows a performance close to the maximum attainable. Some particular implementations, or SDumont computer nodes, required different compiler or library versions. There is also available the PGI *Portland Group Inc.* suite of compilers, but it was not considered in this work.

Two Python distributions are available in the SDumont computing environment: Python versions 3.6.8 and 3.7.7 of the Intel PSXE 2019/2020, and Python versions 3.7.3, 3.8.5, and 3.9.4 of the Anaconda 2018.12 and 2020.11 distribution. Additionally, the Conda environment and package manager was also employed. As already mentioned, some particular implementations, or SDumont computer nodes, required different compiler or library versions.

4.1.2 Compiler evaluation for the Stencil test case

A preliminary performance test for the Stencil test case was performed in order to choose one of the F90 compilers (GNU or Intel) to be adopted for this work. These codes are compiled *Ahead Of Time* (AOT), that is, at compile time. This test also included a comparison between the Intel and Anaconda Python distributions, using the Numba compiler (not Intel or GNU) which is compatible with a subset of Python and NumPy, is a JIT compiler (*Just In Time*), or that is, it compiles at runtime, and does not require major changes to Python code. For Numba, only the compute-intensive kernel is JIT-compiled and executed as machine code, with the rest of the code being interpreted by standard Python. Two Python distributions were compared: Anaconda 2018.12 with Python 3.7.3 and OpenMPI 4.0.1, and Intel PSXE 2019 with Python 3.6.8 and Intel MPI. Two different SDumont nodes are used, B710 and Seq-X.

[2]`http://sdumont.lncc.br/support_manual.php?pg=support#6.8`

Table 4.1 shows the serial and parallel elapsed times for the Intel/GNU F90-compiled implementations and for the Intel/Anaconda implementations executed in the B710 or Seq-X nodes. Running up to 16 processes required a single B710 node, while 36 processes on the B710 required two nodes, and on Seq-X required a single node.

Table 4.1 - Serial and parallel elapsed times (seconds) (Stencil test case, B710 or Seq-X nodes) as a function of the number of processes, for the different F90 and Numba implementations. Best times are highlighted in **red** for Seq-X, and in **blue** for B710.

Implemen-tation	**Serial**	**Number of MPI processes**				
		1	**4**	**9**	**16**	**36**
			F90			
Seq-X/GNU	15.82	**15.64**	4.13	2.09	**1.48**	**1.21**
Seq-X/Intel	**15.59**	15.76	**4.02**	**2.05**	1.58	1.51
B710/GNU	**19.25**	21.91	7.34	6.15	4.68	2.13
B710/Intel	21.87	**20.66**	**7.32**	6.21	4.63	2.20
			Numba			
Seq-X/Anaconda	17.13	17.08	22.61	2.62	1.69	1.44
Seq-X/Intel	17.11	17.95	22.71	2.47	1.78	1.74
B710/Anaconda	30.48	30.53	8.18	6.33	5.86	3.22
B710/Intel	30.37	30.57	8.11	**4.37**	**3.35**	**1.92**

Source: Author's production.

Considering these processing performance results, the GNU compiler suite was adopted for the remaining of this work, except for the profiling tests, shown in the Chapter 5, which required the Intel compiler suite. These results also compare Numba performance for Intel and Anaconda distributions, but Numba is just one of the available Python HPC approaches. In this work, the Python Anaconda distribution was adopted due to the wide availability of libraries and documentation.

4.2 Stencil test case processing performance

This section shows the processing performance of the Stencil test case, for both serial and parallel implementations performed on CPUs/cores of one or multiple core processors of one or more computer nodes, and also on GPUs. The compute-intensive part of the implementations was hand coded, thus not using a specific off-the-shelf external library.

Table 4.2 shows processing times of the test case for the different implementations in one or more SDumont B710 computer nodes. 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. Figure 4.1 shows the processing times as a function of the number of MPI processes for the different implementations, Figure 4.2 shows the corresponding speedups, and Figure 4.3 shows the parallel efficiencies.

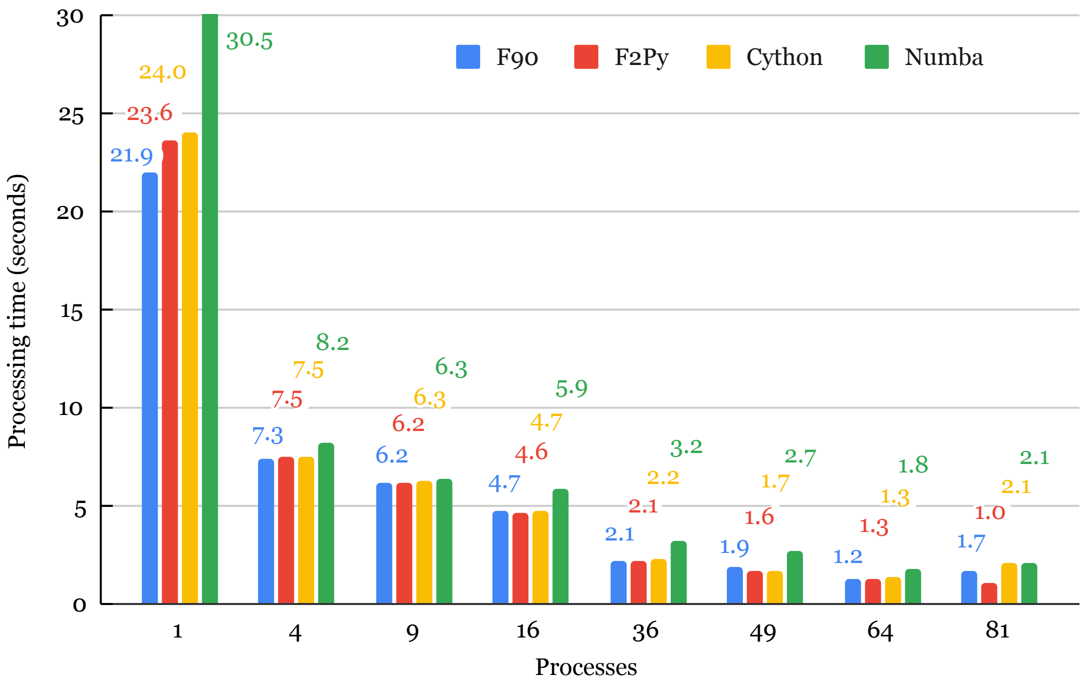
In general, according to Table 4.2, the F90 and the F2PY achieved the best performance, with the latter yielding the lowest processing time of 1.01 s with 81 MPI processes. They are followed by the Cython and Numba implementations, with standard Python well behind. F2PY required little changes to the F90 original code, while Cython and Numba, little changes to the Python code. 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.

Table 4.2 - Performance (Stencil test case, B710 nodes) of the different implementations, depending on the number of MPI processes: processing times, speedups, and parallel efficiencies. Best values for serial or for each number of MPI processes are highlighted in red. The execution time of the compiled serial code was taken as a reference for the calculation of speedup (highlighted in blue).

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

Source: Author's production.

Figure 4.1 - Processing times (seconds) (Stencil test case, B710 nodes) of the different implementations, depending on the number of MPI processes. For convenience, times above 30 s are not fully depicted.



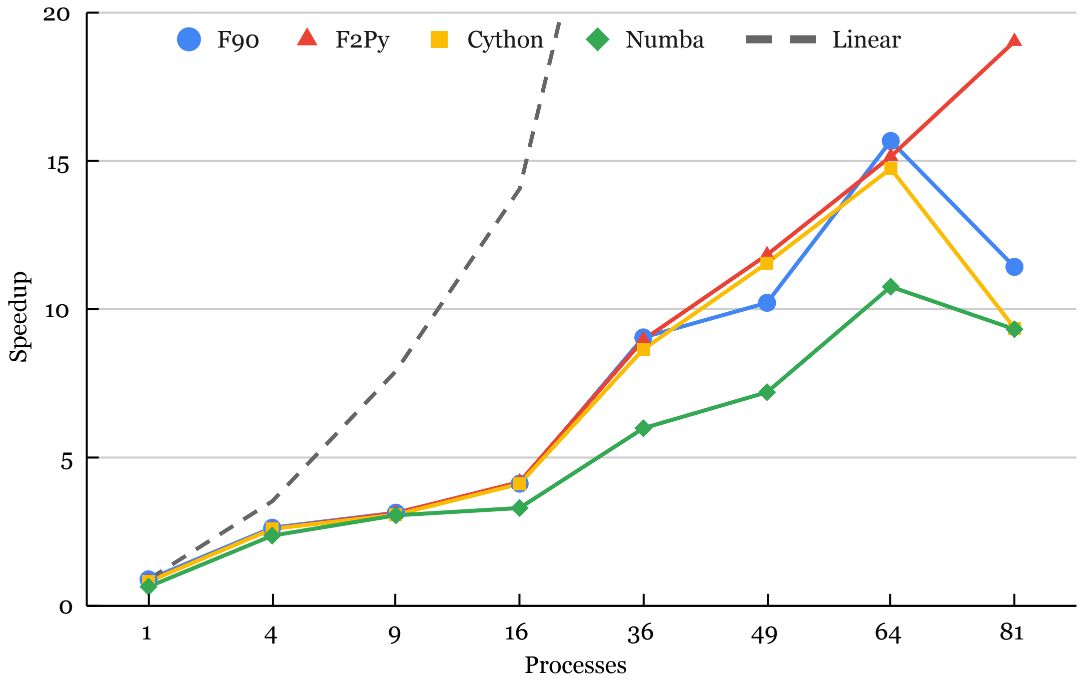
Source: Author’s production.

As can be seen from Table 4.2, Figure 4.1, Figure 4.2 and Figure 4.3, parallel scalability is not good as the test case algorithm updates all grid points at each time step, thus requiring the exchange of boundary grid point temperatures between neighboring subdomains in order to update the corresponding ghost zones. This update implies in communication between processes that compromises the parallel efficiency (below 40% for 9 MPI processes or more). It can be seen that for up to 36 MPI processes, executed in two B710 nodes, all implementations performed similarly, except for standard Python. In the case of 81 MPI processes, executed in 4 computer nodes, the performance of all implementations was significantly lower in comparison to 64 MPI processes, executed in 3 computer nodes. The exception was F2PY that obtained the lowest time with 81 MPI processes.

4.2.1 F90 serial and parallel (CPU)

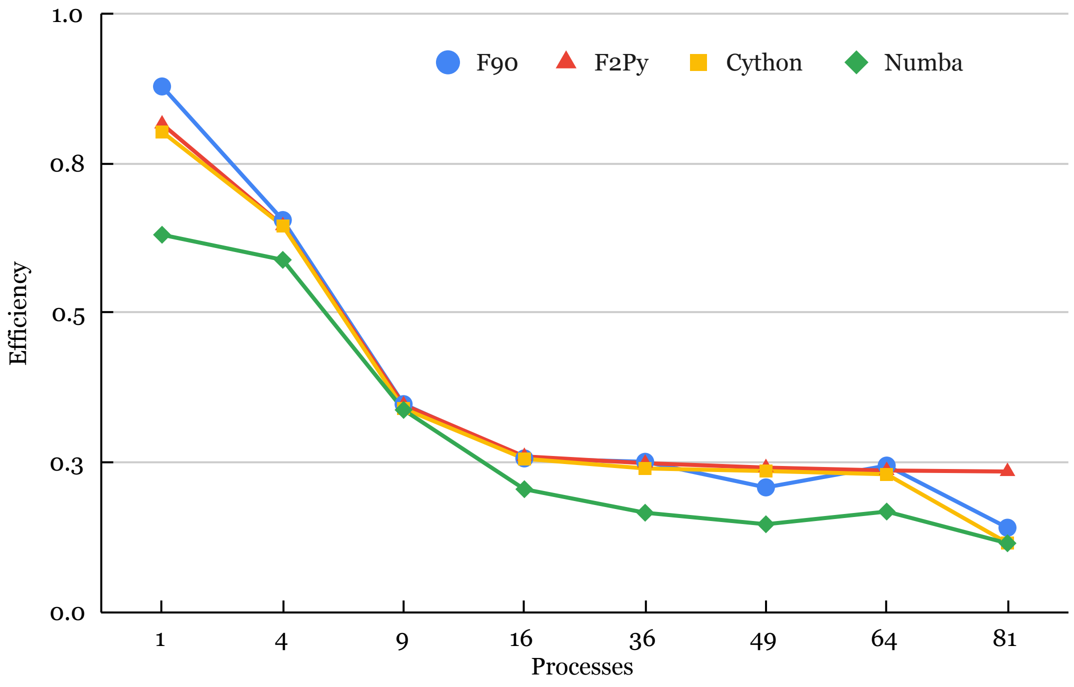
In this work, the parallelization was performed by dividing the domain of the test case into square subdomains of up to 9×9 that are performed by 81 processes. The parameters of the test case are the number of points of the grid, the energy to be inserted, and the number of iterations. F90 obtained the best serial execution

Figure 4.2 - Speedups (Stencil test case, B710 nodes) of the different implementations, depending on the number of MPI processes. The execution time of the *F90 serial* was taken as the reference for speedup calculation. Dashed line denotes the linear speedup.



Source: Author's production.

Figure 4.3 - Parallel efficiencies (Stencil test case, B710 nodes) of the different implementations, depending on the number of MPI processes.



Source: Author's production.

among the implementations, but not for the parallel executions with 16, 49, and 81 processes.

The results were evaluated in the JupyterLab interactive environment using a set of commands called *cell magics* such as *%%writefile* to write the source code to disk, and *%%bash* to access the *shell*, for instance, to build the executable file that will be specified in the Slurm script. Output files resulting from the execution can be read and analyzed on the Notebook, allowing documentation and, consequently, reproducibility.

4.2.2 F2PY serial and parallel (CPU)

F2PY and 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).

4.2.3 Standard Python serial and parallel (CPU)

The standard implementation in Python only uses, as external library, NumPy. It preserves most of the original code, being the 2D compute-intensive loops are performed by NumPy. However, the execution is slow, since it is interpreted. Such implementation serves to develop a proof of concept making use of the Python rapid prototyping and portability, requiring only on a standard Python interpreter. In a further step, the compute-intensive parts of the code can gradually be optimized, taking advantage of the interactive and experimental nature of Python.

4.2.4 Cython serial and parallel (CPU)

The performance of the serial Cython implementation is between F2PY and Numba, and the parallel is close to the ones of F90 and F2PY, from 4 to 64 processes. The serial and parallel versions are based on code that is compiled by Cython. The compute-intensive parts are encapsulated and a Python library is created. This external library is then called from standard Python code. Parallelization is provided by the mpi4py library. In general, Cython is a good choice when F90 code doesn't exist, and we want to use a language relatively close to Python to benefit from readability and maintainability, as well as the fast, iterative development cycle.

4.2.5 Numba serial and parallel (CPU)

Numba's performance was below F90, F2PY, and Cython, but well above standard Python one. 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 compute-intensive core is placed in a function decorated 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 it is intended to make few changes to the code. If applicable, Numba has also the advantage of providing support for GPU execution. Furthermore, since Numba can be JIT-compiled, the machine code is eventually optimized in execution time for a specific architecture, providing portability without the need of a previous AOT compilation.

4.2.6 Numba-GPU

This section intends to compare the single-node performance of: (i) the serial and parallel versions of the Numba-GPU implementation, running on one or more processor cores (CPU) and one or more GPUs, on B715 nodes or Seq-X nodes; (ii) the serial and parallel versions of the F90 implementation, running on one or more processor cores (CPU) on B715 nodes or Seq-X nodes, without GPU. The Seq-X node is an upgraded computer node with newer processors and GPUs, compared to the B715 node. In addition, the Seq-X node has four GPUs.

The Numba-GPU implementation was tested on a B715 node using a single CPU/core and a Tesla K40 GPU with an execution time of 9.35 s, which is half the execution time of the F90 serial implementation (Table 4.3). The same Numba-GPU implementation running on a Seq-X node using a single CPU/core and a single Volta V100 GPU only spent 2.25 s, achieving a speedup of 8.57, which is slightly better than the 9-process MPI F90 implementation on a Seq-X node. The Numba-GPU implementation required major code modifications, compared to standard serial Python code.

The compute-intensive part of the code was encapsulated into a function with a decorator for Numba JIT-compilation and execution in GPU. The block size of $256 = 16 \times 16$ threads has been trial and error optimized for running on V100 (Seq-X node) and K40 (B715 node) GPUs. The remaining code of the implementation, which

Table 4.3 - Performance (Stencil test case, Seq-X and B715 nodes) as a function of the number of processes for the F90 and Numba-GPU implementations. The execution time of the F90-compiled serial code in the B715 node was taken as the reference for the speedup calculation (highlighted in **blue**). Best values for serial or for each number of MPI processes are highlighted in **red**.

Implemen-		**Number of MPI processes**				
tation	**Serial**	**1**	**4**	**9**	**16**	**36**
		Processing time (seconds)				
F90/B715	**19.25**	21.91	7.34	6.15	4.68	2.13
F90/Seq-X	15.82	**15.64**	**4.13**	**2.09**	**1.48**	**1.21**
Numba-GPU/B715	9.35	104.82	27.25	11.71	7.92	4.35
Numba-GPU/Seq-X	**2.25**	49.54	15.16	6.91	6.72	9.09
		Speedup				
F90/B715	1.00	0.88	2.62	3.13	4.11	9.04
F90/Seq-X	1.22	**1.23**	**4.66**	**9.22**	**13.02**	**15.94**
Numba-GPU/B715	2.06	0.18	0.71	1.64	2.43	4.43
Numba-GPU/Seq-X	**8.57**	0.39	1.27	2.79	2.87	2.12
		Parallel efficiency				
F90/B715	1.00	0.88	0.66	0.35	0.26	0.25
F90/Seq-X	1.22	**1.23**	**1.17**	**1.02**	**0.81**	**0.44**
Numba-GPU/B715	2.06	0.18	0.18	0.18	0.15	0.12
Numba-GPU/Seq-X	**8.57**	0.39	0.32	0.31	0.18	0.06

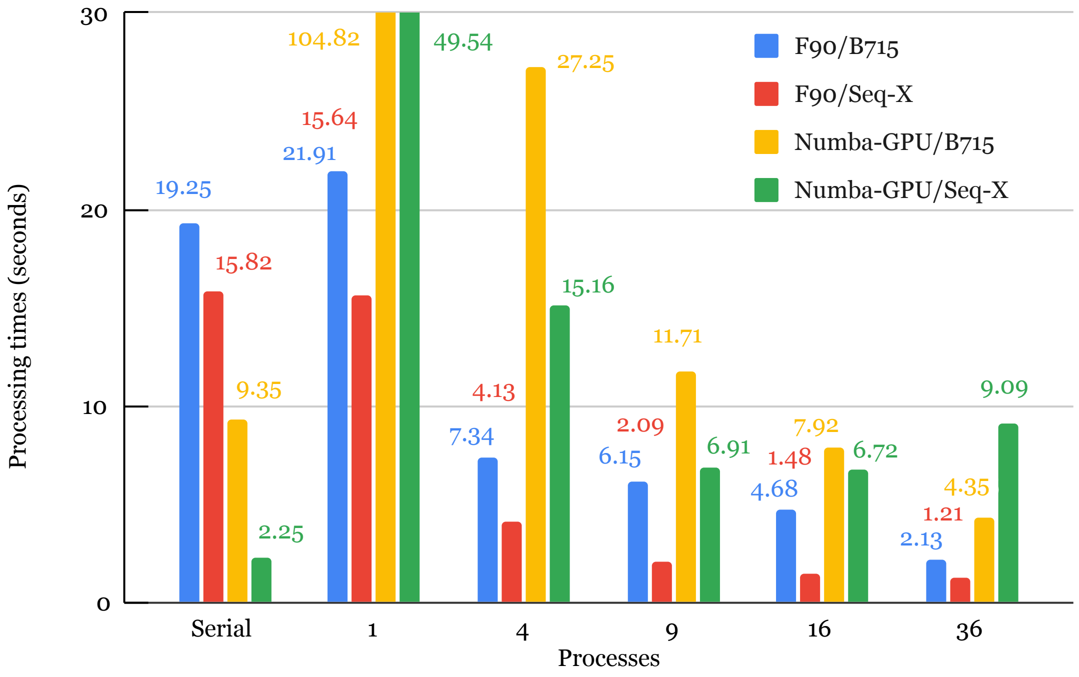
Source: Author's production.

is not compute-intensive, was executed in an interpreted form by Python using the CPU.

The performances of the F90 serial and parallel implementations for CPU, and the Numba-GPU implementation for a B715 node or a Seq-X node, are shown in Table 4.3 and in Figure 4.4, Figure 4.5, and Figure 4.6. In the parallel execution of the F90 implementation on the 24-core B715 node, for up to 16 MPI processes a single node was used, and for 36 MPI processes two nodes were used.

Parallel execution of Numba-GPU on B715 nodes was done using 2 MPI processes per node. Each process is executed in a single core of a processor, leaving the remaining 11 cores of the processor in idle, and each process uses one GPU. Therefore, execution with 36 MPI processes demanded 18 nodes B715. However, Numba-GPU parallel execution in Seq-X nodes was restricted to a single Seq-X node with 48 cores and 4 GPUs. Thus, for 1 to 4 MPI processes, each process used an exclusive GPU, while for 9, 16, and 36 processes, GPUs are assigned to processes in a round-robin distribution,

Figure 4.4 - Processing times (seconds) (Stencil test case, Seq-X and B715 nodes) as a function of the number of MPI processes for the F90 and Numba-GPU implementations executed on the Seq-X (**red** and **green** bars) and on the B715 node (**blue** and **orange** bars). For convenience, times above 30 s are not fully depicted.

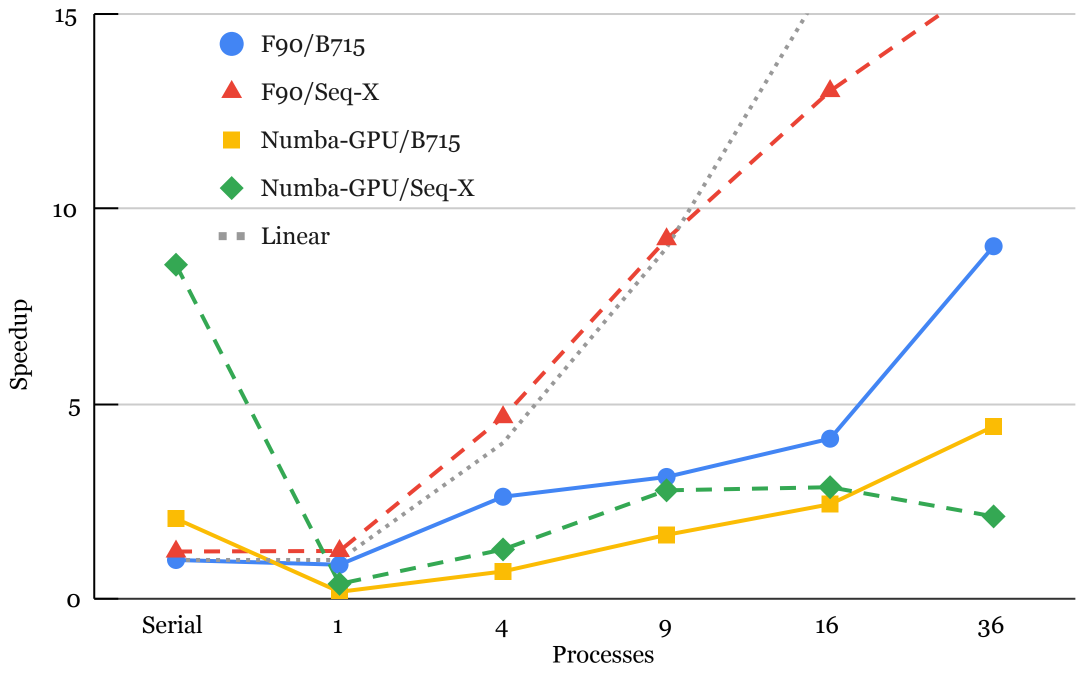


Source: Author's production.

compromising the performance. A further improvement to take advantage of Numba ease of programming would be to use a single Seq-X node using 48 MPI processes, assigning 4 processes to be executed using Numba-GPU, leaving the remaining 44 processes executed using CPU (Numba-CPU). Another improvement, assuming many Seq-X nodes available, would be to use 4 MPI processes per node, similarly to the B715 Numba-GPU parallel implementations.

Table 4.3 and Figure 4.4 shows the processing times (seconds) as a function of the number of MPI processes, for the F90 and Numba-GPU implementations executed on B715 and Seq-X nodes. In this table, F90 implementation execution times in nodes B715 were extracted from Table 4.2. The Numba-GPU implementation was executed on B715 nodes with 2 MPI processes per node using 2 GPUs, and thus each process is executed on a core/CPU paired with a GPU. Therefore, 1 MPI process uses 1 node, 4 processes uses 2 nodes, 9 processes uses 5 nodes, 16 processes uses 8 nodes, and 36 processes uses 18 nodes. The Numba-GPU implementation was executed on a single Seq-X node with up to 36 processes using 4 GPUs. Therefore, for 1 to 4 MPI processes, each process used an exclusive GPU, while for 9, 16, and 36 processes,

Figure 4.5 - Speedup (Stencil test case, Seq-X and B715 nodes) as a function of the number of MPI processes, for the F90 and Numba-GPU implementations, serial and parallel versions, executed on the Seq-X (dashed lines) and on the B715 (solid lines). The execution time of serial implementation F90 on B715 was taken as reference for speedup calculation. The dotted line gray is the linear speedup.



Source: Author's production.

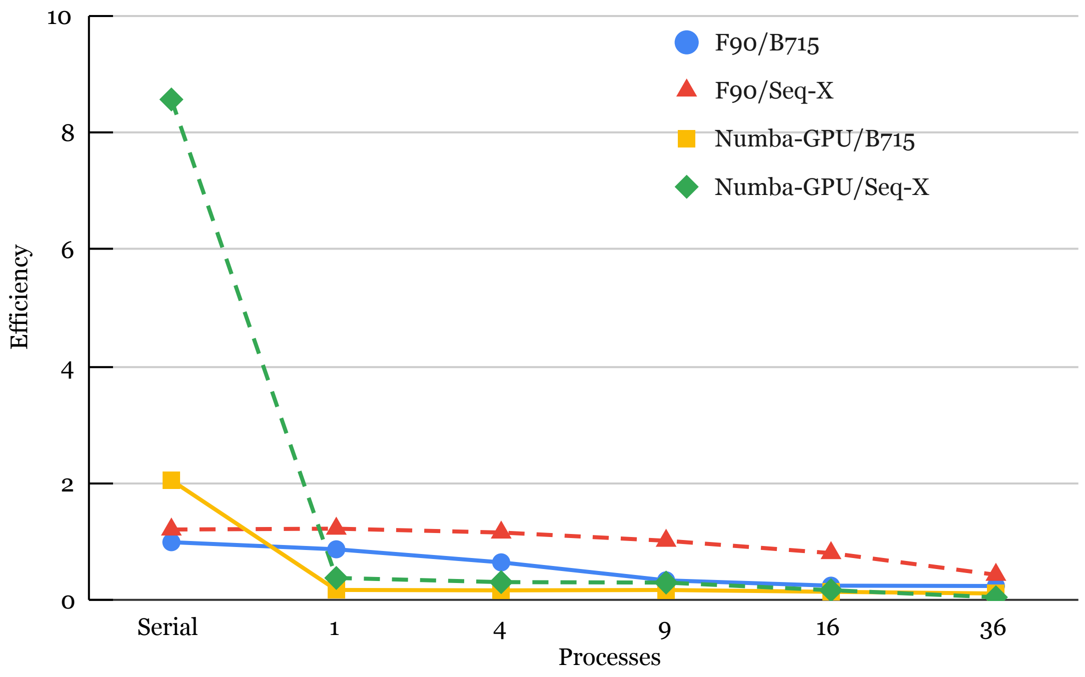
GPUs are assigned to processes in a round-robin distribution, compromising the performance.

Figure 4.5 shows the Speedup as a function of the number of MPI processes, for the serial and parallel versions of the F90 and Numba-GPU implementations, executed on Seq-X and B715 nodes. The dotted line represents the linear speedup. Two points should be highlighted: the speedup of 8.57 of the Numba-GPU serial version on the Seq-X node, and the good scalability of F90 parallel on the Seq-X node up to 9 MPI processes. Obviously, the speedup of serial versions of Numba-GPU is only due to GPUs.

Figure 4.6 shows the parallel efficiencies as a function of the number of MPI processes, for the serial and parallel F90 and Numba-GPU implementations, executed on the Seq-X node and nodes B715. In general, these implementations have very low parallel efficiency.

Table 4.4 shows the execution times of some parts of the code as a function of the

Figure 4.6 - Parallel efficiencies (Stencil test case, Seq-X and B715 nodes) as a function of the number of MPI processes, for the F90 and Numba-GPU implementations, serial and parallel versions, executed on the Seq-X node (dashed lines) and on the B715 nodes (solid lines).



Source: Author's production.

number of MPI processes, for the Numba-GPU implementation executed on Seq-X and B715 nodes. All these times were measured in the Python code using the native Python wall time function. The three selected parts are executed every time step: *Kernel*, *Memory transfer* and *Insertion of energy*.

The *Kernel* is the most computationally intensive part of the code, and as it is JIT-compiled, it is slow on the first run because of the build time, and a little faster on later runs because the machine code is stored in memory requiring no additional compilation. *Memory transfer* time measures the 2D array transfer time between host memory and device memory, and vice-versa, at each time step, which accounts for most of the total execution time in the MPI implementations. Please note that such transfer at every time step only exists for the MPI version, even using 1 process, but in the serial version, the 2D array is transferred to the GPU in the first time step, being the remaining time steps performed in the GPU without any transfer, except for the last time step. This explains the huge difference between the transfer times of the serial and 1-process MPI executions.

This Numba-GPU implementation for the Stencil test case shows that writing HPC

Table 4.4 - Time elapsed (seconds) (Stencil test case, Seq-X and B715 nodes) for selected code snippets, as a function of the number of MPI processes, for the Numba-GPU implementation.

Implemen-		**Number of MPI processes**				
tation	**Serial**	**1**	**4**	**9**	**16**	**36**
Numba-GPU/B715	9.35	104.82	27.25	11.71	7.92	4.35
Kernel	1.08	1.77	1.34	1.01	0.84	1.19
Mem transf	0.32	102.97	25.51	9.73	5.48	2.75
Insert energy	7.93	0.04	0.04	0.02	0.00	0.00
Numba-GPU/Seq-X	2.25	49.54	15.16	6.91	6.72	9.09
Kernel	0.28	0.86	0.27	0.31	0.40	0.46
Mem transf	0.27	48.63	14.55	5.78	4.18	6.69
Insert energy	1.67	0.02	0.02	0.01	0.00	0.00

Source: Author's production.

code for GPU execution takes some effort, but generally brings some performance, as shown by the serial GPU implementation running on Seq-X node, being 8 times faster than the Serial CPU implementation. However, the performance of the parallel implementation show that it demands further optimizations in order to minimize the overheads of MPI communication and memory transfers between GPU and host, and thus explore the processing power of the GPU.

4.3 FFT test case processing performance

This section shows the processing performance of implementations of the Fast Fourier transform test case, running on CPUs, i.e., on one or multiple processor cores of one or more computer nodes, and on GPUs, including an implementation optimized for NUMA. The compute-intensive part of the implementations uses out-of-the-box external libraries, differently from the previous test case.

Table 4.5 shows processing times of the test case for the different implementations in one or more SDumont B710 computer nodes. 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. According to the same table, the F90 implementation achieved the best performance, followed by the F2PY implementation. The shortest time was obtained by the F90 implementation with 48 MPI processes (2.22 s) processes. Numba performed generally better than Cython and Python, but worse than F2PY and F90. The very low performance of the standard Python serial and parallel versions shows the convenience of using other Python-compatible implementations like F2PY or Numba.

Table 4.5 - Performance (FFT test case, B710 nodes) of the different implementations, depending on the number of MPI processes: processing time, speedup, and parallel efficiencies. Best values are highlighted in **red**. The execution time of the *F90 serial* implementation was taken as the reference for speedup calculation and is shown in **blue**.

Implemen-tation	**Serial**	**Number of MPI processes** 1	**4**	**16**	**24**	**48**	**72**	**96**
		Processing time (seconds)						
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

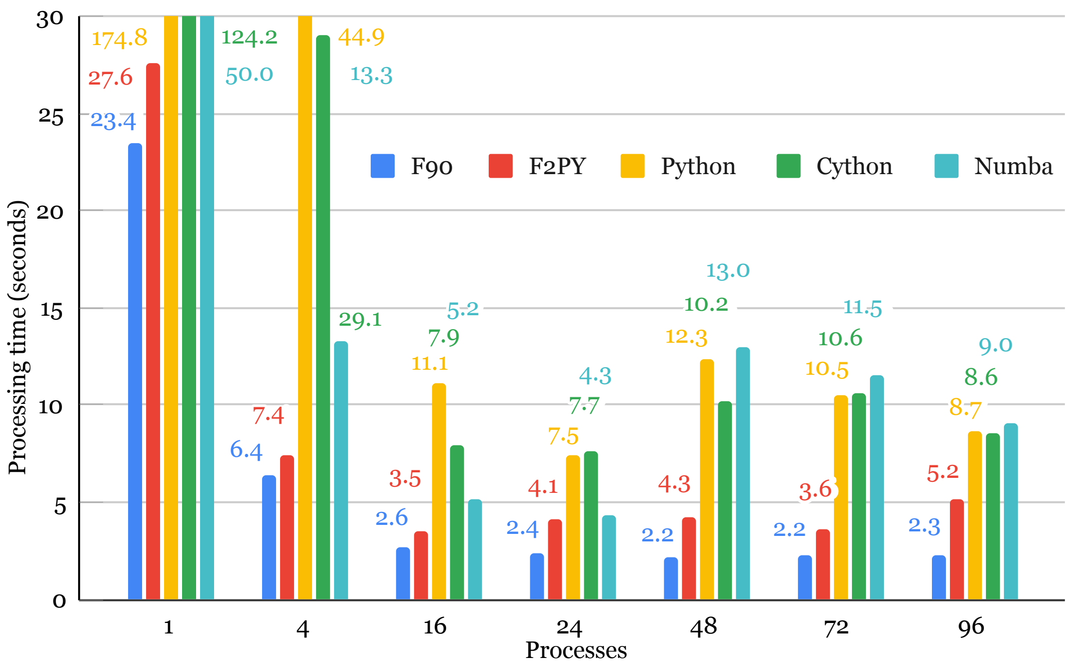
Source: Author's production.

Figure 4.7 shows the processing times as a function of the number of MPI processes for the different implementations, Figure 4.8 shows the corresponding speedups calculated using as reference to the time of the F90 serial implementation, and Figure 4.9 shows the parallel efficiencies. It can be seen that the parallel scalability is poor for all implementations due to the MPI communication. Only the F90 and F2PY implementations show some parallel scalability, and above 4 MPI processes all implementations presented efficiencies below 50%.

4.3.1 F90 serial and parallel (CPU)

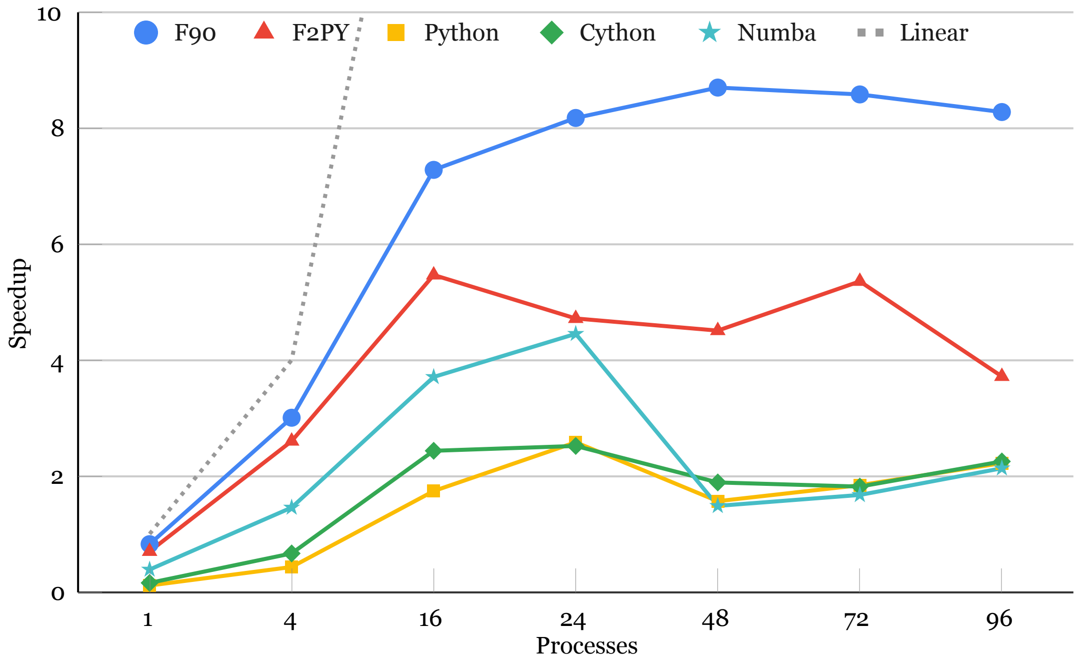
The F90 serial and parallel implementations presented the best performance, but with poor parallel scalability, except for 4 processes, with efficiency of 75%, reaching for 96 processes an efficiency of only 9%.

Figure 4.7 - Processing times (seconds) (FFT test case, B710 nodes) of the different implementations as a function of the number of MPI processes. For convenience, times above 30 s are not fully depicted.



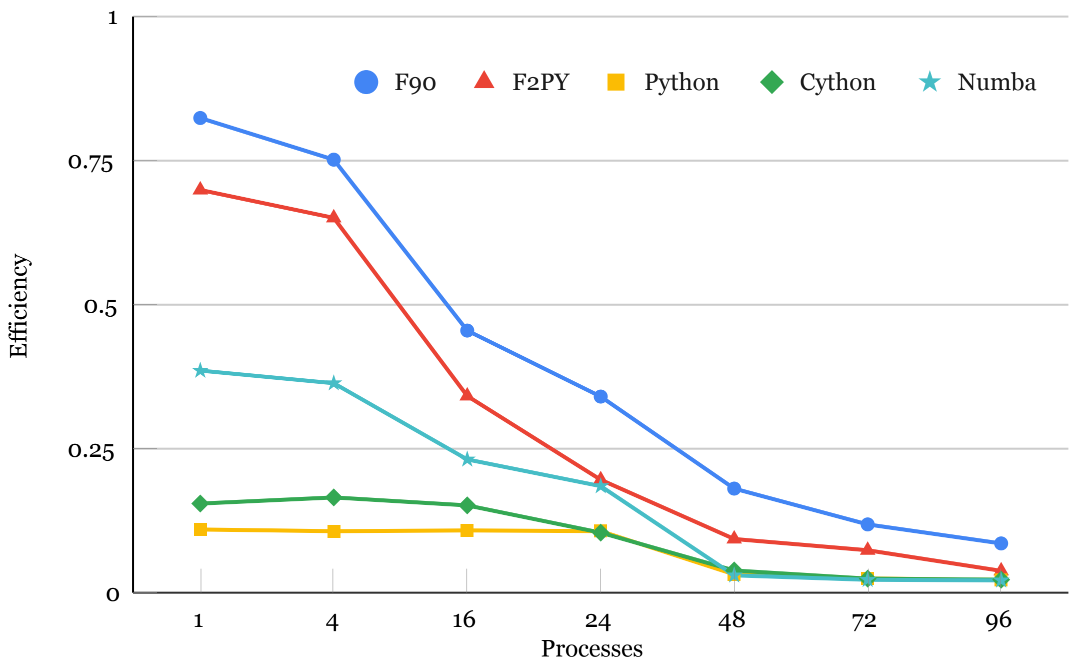
Source: Author's production.

Figure 4.8 - Speedups (FFT test case, B710 nodes) of the implementations as a function of the number of MPI processes. Dotted lines denote linear speedup.



Source: Author's production.

Figure 4.9 - Parallel efficiencies (FFT test case, B710 nodes) as a function of the number of MPI processes depending on the number of MPI processes.



Source: Author's production.

4.3.2 F2PY serial and parallel (CPU)

F2PY serial and parallel implementations performed slightly worse than F90. Parallel scalability was also poor, except for 4 processes, with efficiency of 65%, reaching for 96 processes an efficiency of only 4%.

4.3.3 Standard Python serial and parallel (CPU)

Similarly to other test cases, the performance of the standard Python serial and parallel implementations is very poor, but these implementations serve as a starting point to execute making use of Python features, such as portability to execute in different computing environments. In a further step, compute-intensive parts of the code can be replaced by an optimized library, using for instance F2PY, Cython or Numba, as discussed in the next sections.

4.3.4 Cython serial and parallel (CPU)

In general, the performances of the Cython and Python implementations are the worst, for both serial and parallel implementations. Serial times are high and parallel scalability is very poor. The serial Cython implementation uses some native extensions, pyFFTW library, and creates a Python library which is then used in a main Python code that imports the library and calls the new function.

4.3.5 Numba serial and parallel (CPU)

In general, Numba implementation performance was worse than the F90 and F2PY ones, but better than the Python and Cython implementations. The lower time of Numba was 4.33 s for 24 MPI processes, relatively close to F2PY implementation.

4.3.6 CuPy (GPU)

Similarly to the tests performed with the Numba-GPU for the Stencil test case in Subsection 4.2.6, this section intends to compare the single-node performance of the serial CuPy implementation, using a single GPU on a B715 or a Seq-X node, to the F90 serial and parallel implementations, up to 24 processes on a B715 or a Seq-X node, but with no GPU. Performance results for both approaches are presented in Table 4.6. Please note that the F90 implementations uses the FFTW library, while the Python one uses the fftn routine of the CuPy library. The same F90 and CuPy implementations were executed in both B715 and Seq-X nodes, but the latter node is newer, and thus deliver more processing performance.

The CuPy serial implementations (1 processor core and 1 GPU) had a poor performance, compared to the F90 parallel implementations executed with 4 to 24 processes on both the B715 or the Seq-X node. In addition, the parallel F90 implementation executed had acceptable parallel scalability, similar to when it was performed in both nodes, if considered as reference the F90 serial time on the SEQ-X node (12.74 s). As an example, for 24 processes, efficiencies were 40% and 55%, executed in nodes B715 and Seq-X, respectively. The corresponding graphs of Table 4.6 are shown in Table 4.6 and Figure 4.10, Figure 4.11 and Figure 4.12.

The CuPy implementation for GPU didn't require many code modifications compared to the Python implementation for CPU as CuPy doesn't wrap the code in a function like Numba-GPU. The CuPy library is similar to the NumPy library, being used to copy the 3D array to the device memory (GPU), execute the fftn routine, and calculate the checksum of the array elements.

4.3.7 Optimization for NUMA

In order to check the influence of NUMA optimization (Section 2.16), this section show processing performance using always 16 processes for the different implementations.

Table 4.6 - Performance (FFT test case, Seq-X and B715 nodes) of the serial CuPy implementation and of the F90 implementations, as a function of the number of MPI processes. The execution time of the serial F90 implementation executed in the B715 node was taken as a reference for the calculation of speedup (highlighted in blue).

Implementation	GPU	Serial	Number of MPI processes 1	4	16	24
			Processing time (seconds)			
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**					

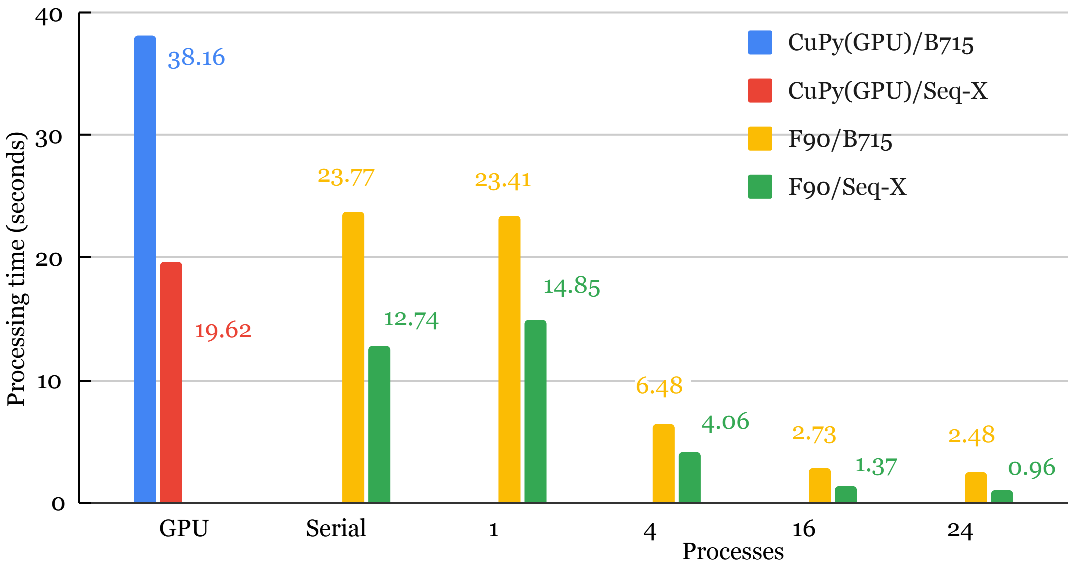
Source: Author's production.

Since executions were performed in 24-core B710 or 48-core Seq-X nodes, such number of processes may be unevenly distributed between the processors. This can be avoided by using the *cpu_bind* option with (for instance) the *distribution=block:cyclic* attribute in the Slurm script, as described in Section 2.16, and referred in this section as "option Bind".

Figure 4.13 shows the processing times (seconds) and Table 4.7 also shows the speedup and parallel efficiency, of the different implementations of the FFT test case, for 16 MPI processes on B710 and Seq-X nodes, with/without NUMA optimization, respectively Bind/None. The Bind option reduced time by 10% in average for node B710, and by 21% in average for node Seq-X. In the latter, Numba had the biggest reduction (32%).

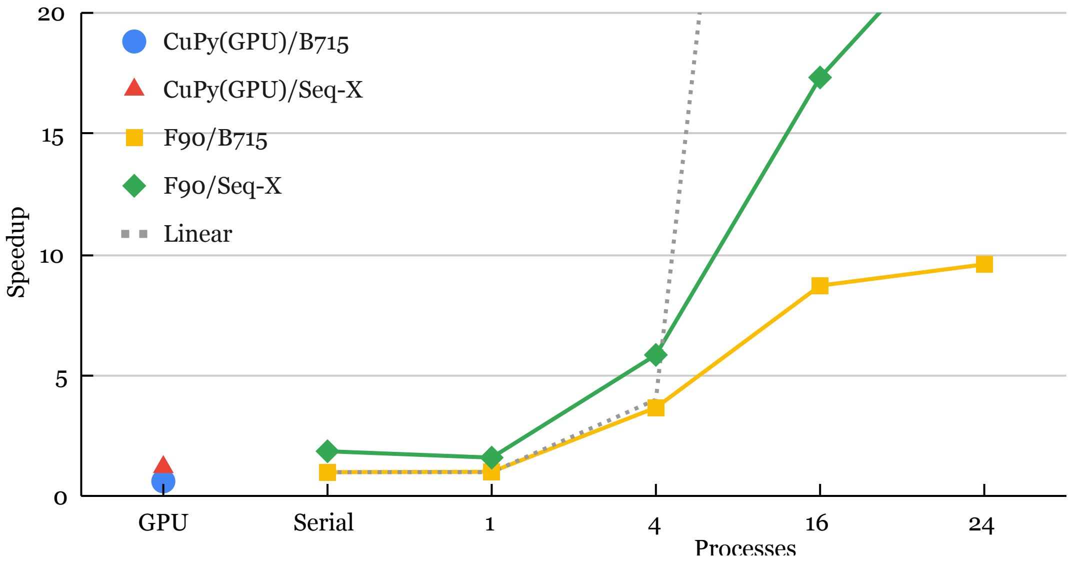
Figure 4.14 shows the speedup of the different FFT test case implementations, speedup and parallel efficiencies for 16 MPI processes, running on nodes B710 and Seq-X with/without NUMA optimization (respectively Bind/None). F90 execution

Figure 4.10 - Processing times (seconds) (FFT test case, Seq-X and B715 nodes) depending on the number of MPI processes for the F90 and CuPy implementations performed on the Seq-X (**red** and **green** bars) and on the B715 node (**blue** and **orange** bars).



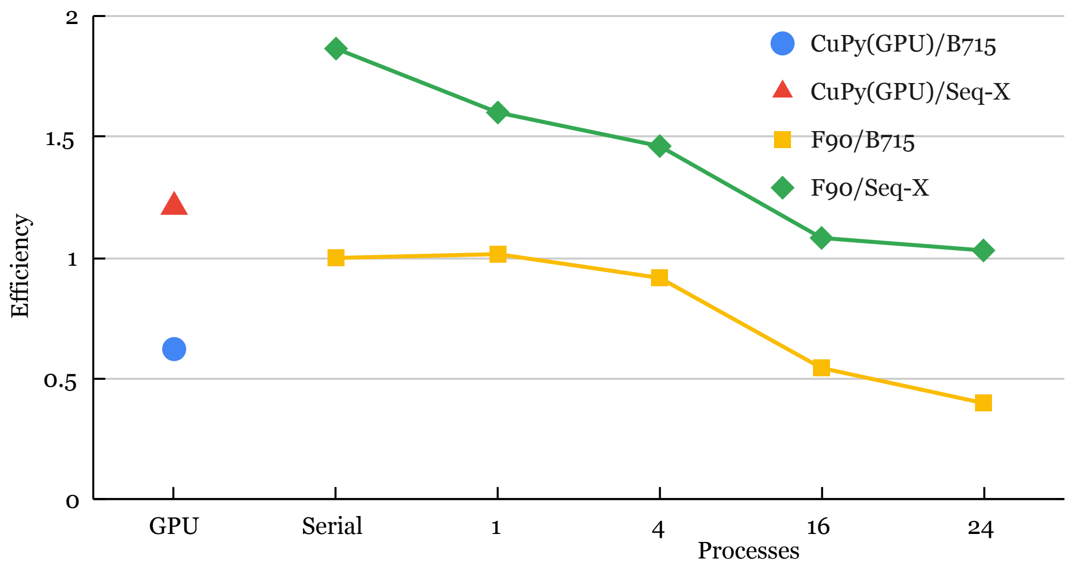
Source: Author's production.

Figure 4.11 - Speedups (FFT test case, Seq-X and B715 nodes) of the serial CuPy implementation and of the F90 implementations, as a function of the number of MPI processes, executed on the Seq-X (**red** triangle and **green** line) and on the B715 node (**blue** dot and **orange** line). The execution time of the serial F90 implementation executed in the B715 node was taken as a reference for the calculation of speedup (highlighted in **blue**). The dotted line indicates the linear speedup.



Source: Author's production.

Figure 4.12 - Parallel efficiencies (FFT test case, Seq-X and B715 nodes) of the serial CuPy implementation and of the F90 implementations, as a function of the number of MPI processes, performed on the Seq-X (red triangle and green line) and on the B715 node (blue dot and orange line).



Source: Author's production.

time with 16 processes without optimization was taken as reference for speedup calculation. Figure 4.15 shows the parallel efficiencies of the different implementations of the FFT test case, speedup and parallel efficiencies for 16 MPI processes running on nodes B710 and Seq-X with/without NUMA optimization (respectively Bind/None).

4.4 Random Forest test case processing performance

This section shows the processing performance of the random forest (RF) test case (Table 4.8), which has the compute-intensive part executed by an external library. Python, Numba, and Cython use the Scikit-learn library (Section 2.1), while F90 and F2PY use the PARF library (Section 2.15). Parallelization is done using the MPI library for F90 and F2PY, or the IPP library, for Python, Cython, and Numba. This section shows the serial and parallel processing performance of the CPU implementations, i.e., executed by one or multiple processor cores of one or more computer nodes, with no GPU.

Table 4.8 shows the processing times for the different implementations on SDumont B710 execution nodes. The same table also shows processing times for the serial and parallel versions with 1, 4, 16, 24, 48, 72, and 96 processes. Execution with 1 to 24 processes use 1 node, 48 processes use 2 nodes, 72 processes use 3 nodes, and 96

Table 4.7 - Processing times (seconds) (FFT test case, B710 and Seq-X nodes), speedups and parallel efficiencies of the different implementations for 16 MPI processes, using or not the Bind option for NUMA optimization. The execution time of F90 without such option (None) on node B710 was taken as reference for speedup calculation (highlighted in blue). Best values are highlighted in red.

Implemen-tation	**Number of MPI processes**			
	B710		**Seq-X**	
	None	**Bind**	**None**	**Bind**
	16	**16**	**16**	**16**
	Processing time (seconds)			
F90	**2.65**	**2.47**	**1.65**	**1.33**
F2PY	3.53	3.20	2.09	1.54
Python	11.09	10.10	6.60	5.66
Cython	7.92	6.52	5.31	4.51
Numba	5.20	4.70	3.40	2.30
	Speedup			
F90	**1.00**	**1.07**	**1.61**	**1.99**
F2PY	0.75	0.83	1.27	1.73
Python	0.24	0.26	0.40	0.47
Cython	0.33	0.41	0.50	0.59
Numba	0.51	0.56	0.78	1.15
	Parallel efficiency			
F90	**0.06**	**0.07**	**0.10**	**0.12**
F2PY	0.05	0.05	0.08	0.11
Python	0.01	0.02	0.03	0.03
Cython	0.02	0.03	0.03	0.04
Numba	0.03	0.04	0.05	0.07

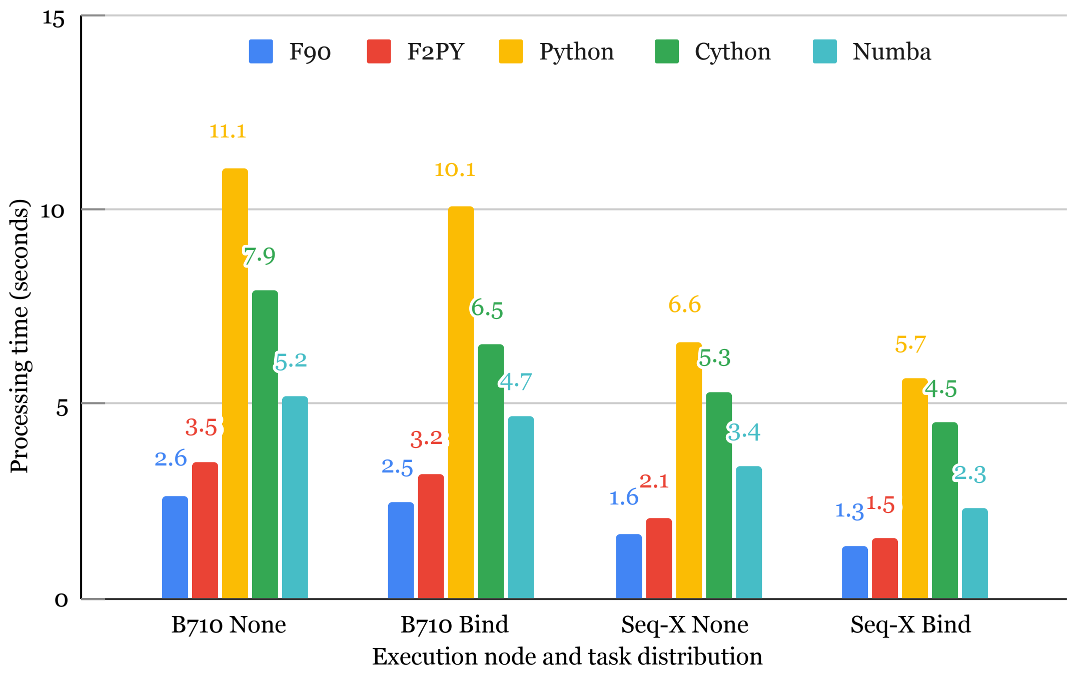
Source: Author's production.

processes use 4 nodes. Figure 4.16 also shows the processing times as a function of the number of processes for the different implementations, Figure 4.17 shows the corresponding speedups, and Figure 4.18 shows the parallel efficiencies.

According to Table 4.8, the lower time was 11 s for the Python implementation with 24 processes, while the higher time was 141.71 s for the parallel F2PY implementation using 1 MPI process. Numba serial obtained a slightly better time than Python serial, however to standardize the analysis, the Python serial time was used as a reference for speedup and efficiency calculations, instead of serial F90, which demanded a higher execution time.

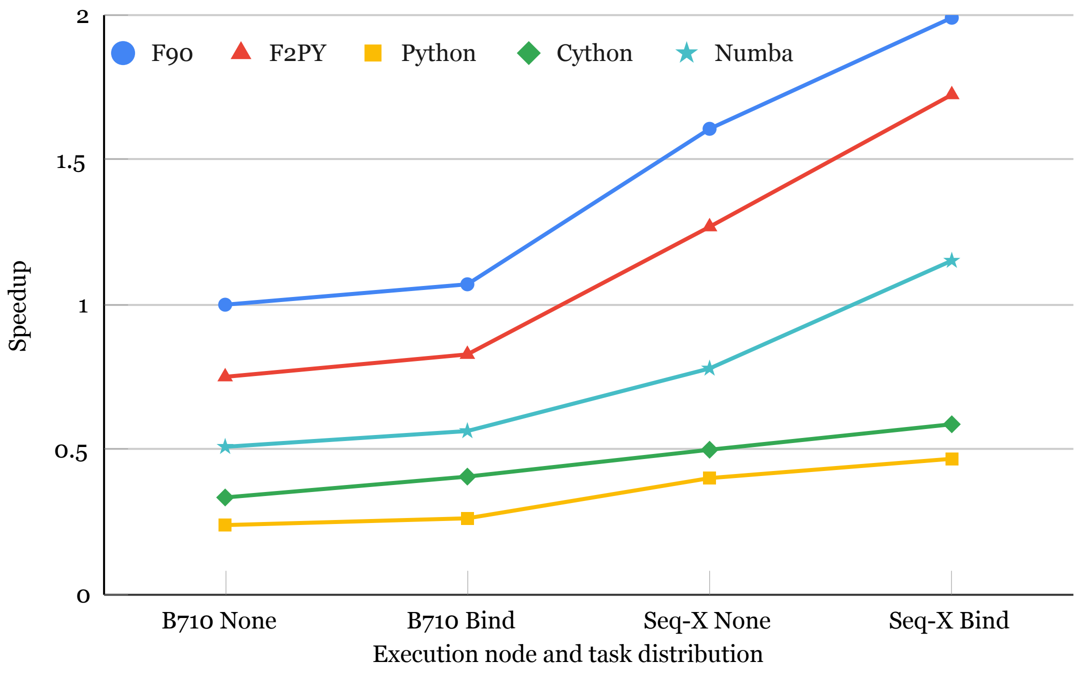
Serial implementations achieved slightly better time than the corresponding 1-process parallel versions, as the latter add the overhead of parallel execution mechanisms

Figure 4.13 - Processing times (seconds) (FFT test case, B710 and Seq-X nodes) of the different implementations for 16 MPI processes with/without NUMA optimization (respectively Bind/None).



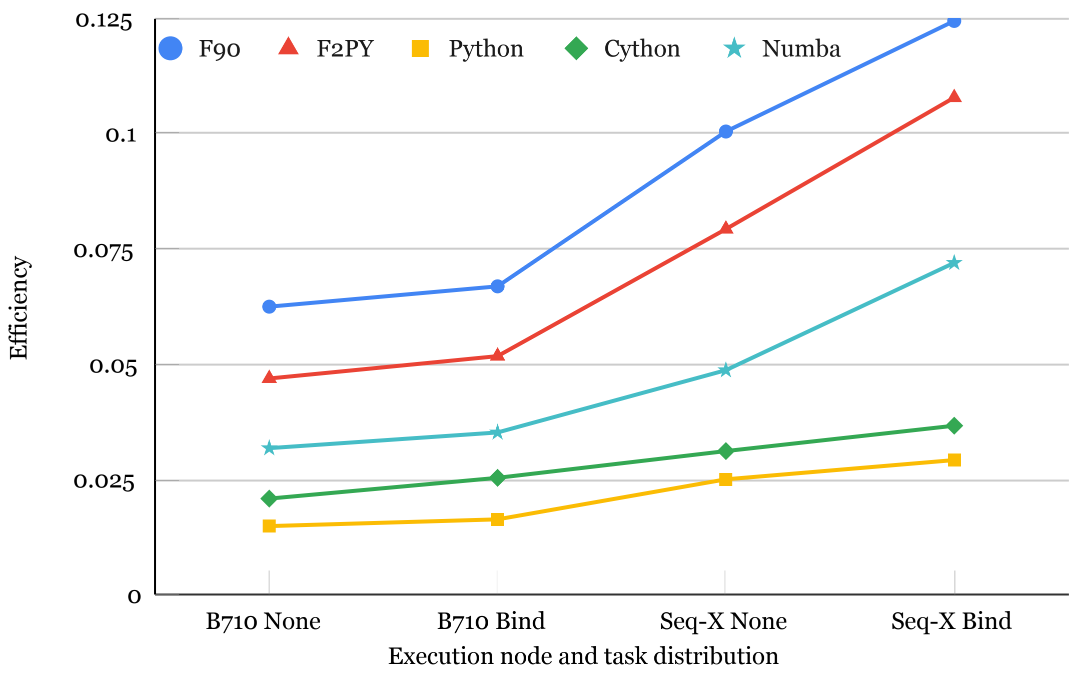
Source: Author's production.

Figure 4.14 - Speedups (FFT test case, B710 and Seq-X nodes) of the different implementations for 16 MPI processes with/without NUMA optimization (respectively Bind/None).



Source: Author's production.

Figure 4.15 - Parallel efficiencies (FFT test case, B710 and Seq-X nodes) of the different implementations for 16 MPI processes, with/without NUMA optimization (respectively Bind/None).



Source: Author's production.

and libraries. In general, the Python implementation have the best performance in all cases, followed by Numba. However, speedups and parallel efficiencies were very low for all implementations.

Since the compute-intensive part of the code is executed by external libraries, just the serial implementations can be used for an initial performance evaluation. The serial F90 and F2PY implementations, which use the PARF library (Section 2.15), are much slower than the Python, Numba or Cython implementations, which use the Scikit-learn library. Another aspect is that Numba or Cython implementations did not perform better than the standard Python implementation, indicating that a suitable choice of optimized library was more important, and that the use of Numba or Cython resulted in performance overheads.

Speedups and parallel efficiencies were very low for all implementations. As can be seen in Table 4.8, and in Figure 4.16, Figure 4.17, and Figure 4.18, execution times decreased up to 16 (using 1 node), 24 or 48 processes (using 2 nodes), depending on the implementation. Parallel scalability was very poor, since all speedups are below 2.5 and all efficiencies, below 0.5.

Table 4.8 - Performance (Random Forest test case, B710 nodes) of the different implementations, depending on the number of processes: processing times, speedups, and parallel efficiencies. Best values for each case are highlighted in **red**. The execution time of the serial code was taken as a reference for the calculation of speedup, shown in **blue**.

Implementation	**Serial**	**Number of MPI processes** 1	**4**	**16**	**24**	**48**	**72**	**96**
		Processing time (seconds)						
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.

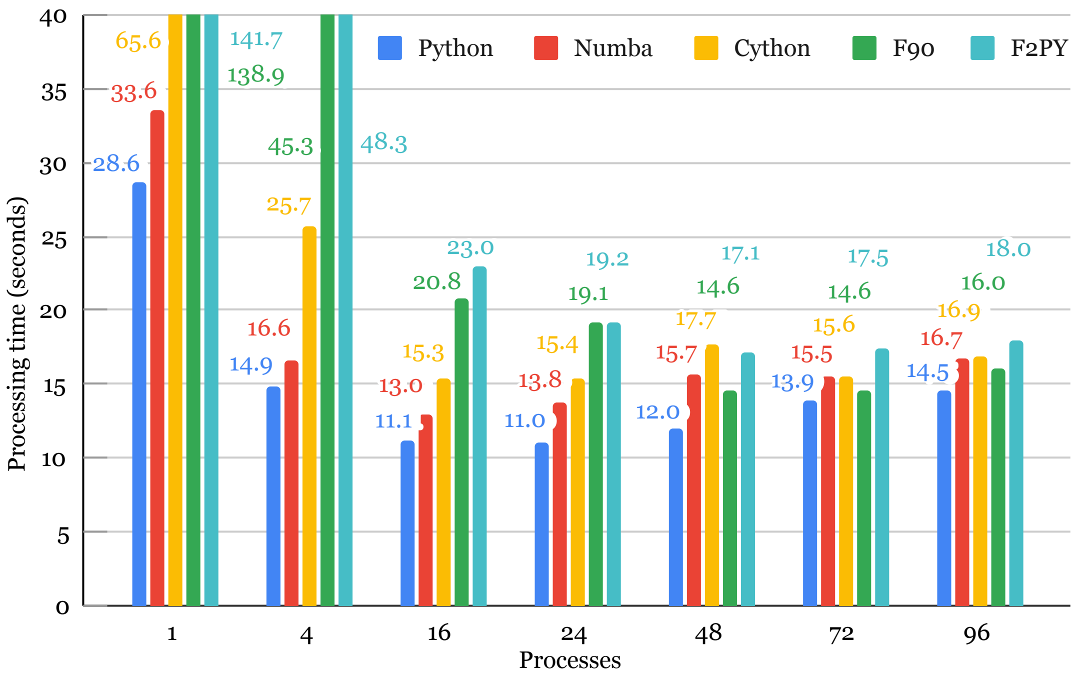
4.4.1 F90 Serial and parallel (CPU)

The F90 implementation requires the PARF library to be configured, and built using the Intel compiler. In the case of the parallel version, it also needs to be configured to use the MPI library. PARF has an interface to the command line, which was used in this implementation. Two different libraries were compiled and generated, one for the sequential version, the other for the parallel version. Once created, the libraries are used directly via command line, using the files containing the datasets as arguments of the library function.

4.4.2 F2PY Serial and parallel (CPU)

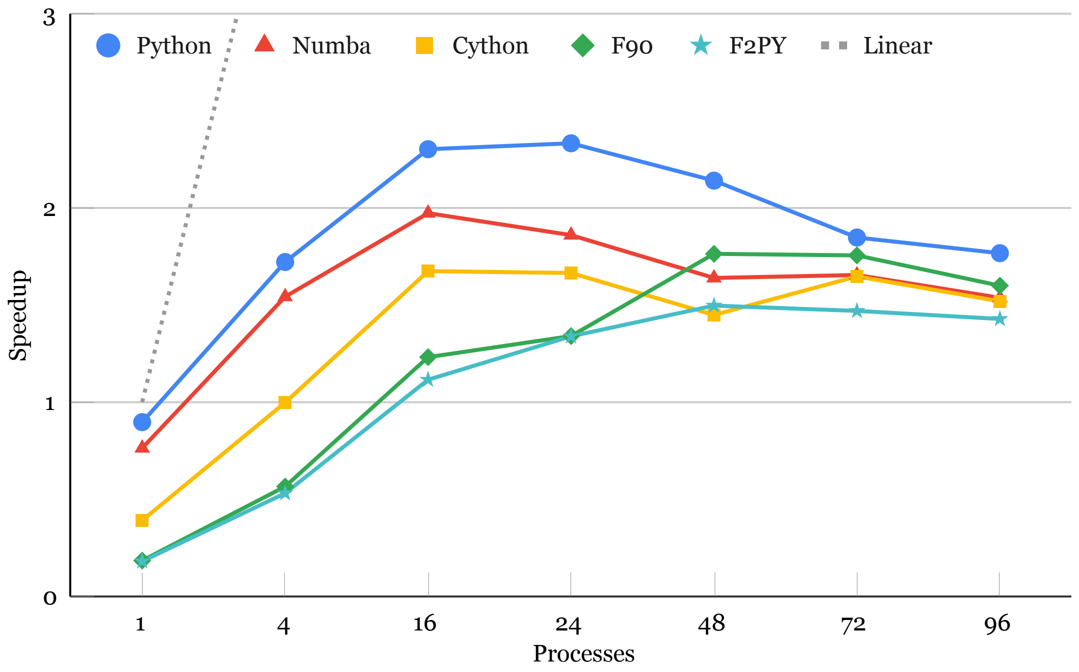
In this implementation, F2PY wrapped the F90 code, which reuses code from the PARF library and executes the compute-intensive part of the code, into a new Python library using F2PY. The remaining Python part of the code is simple,

Figure 4.16 - Processing times (seconds) (Random Forest test case, B710 nodes) for the different implementations as a function of the number of processes. For convenience, times above 40 s are not fully depicted.



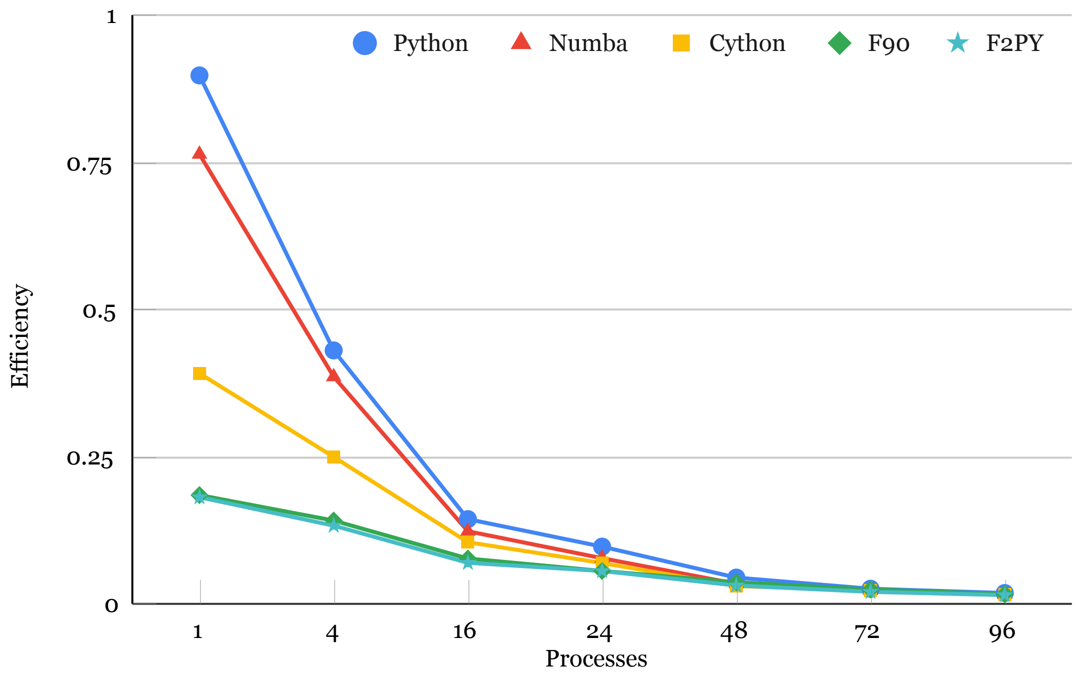
Source: Author's production.

Figure 4.17 - Speedups (Random Forest test case, B710 nodes) of implementations as a function of the number of processes. Serial Python time was used as reference for calculation of the speedup. The dotted line denotes linear speedup.



Source: Author's production.

Figure 4.18 - Parallel efficiencies (Random Forest test case, B710 nodes) of implementations as a function of number of processes.



Source: Author's production.

it loads the library, loads the datasets, calls the library, and displays the result. Overall performance of this F2PY implementations is close to that of F90. The F2PY serial version is slightly faster than the F90 version, probably due to optimizations performed in the original F90 code when compiled by F2PY.

There are no major changes in the parallel version, the F90 source code of the PARF library is configured to use the MPI library, and the new parallel version of the library is built by F2PY. The newly created library is then used in the same Python code as the serial version, being the Slurm configuration file written accordingly for MPI execution.

4.4.3 Standard Python Serial and parallel (CPU)

The standard Python code is more complex than the F90 implementation, which uses the PARF library, since the Scikit-learn library is more general and requires specifying the estimator (Random Forest), the related parameters (for example, the number of trees of the Random Forest), to execute the training phase using the corresponding dataset, and then to perform the test phase using the test dataset. Finally, to calculate the chosen Kappa rank metric. In addition, a small Python code was used to read the original ARFF dataset from disk and to convert it to the

Scikit-learn format. The parallel version is similar, requiring only the addition of a Python line of code specifying the IPP library as the parallel backend in the training phase.

4.4.4 Cython Serial and parallel (CPU)

The Cython implementation is very similar to the standard Python, with the compute-intensive part of the code executed by the Scikit-learn library. Cython is only used for the remaining part of the code, and thus it was not expected a significant performance improvement. However, the use of the Cython adds a small overhead, and performance was worse than that of standard Python. Parallelization was done using the Scikit-learn library with the IPP backend.

4.4.5 Numba Serial and parallel (CPU)

Since the compute-intensive part of the Random Forest code is executed using the Scikit-learn library, Numba JIT-compilation was only applied to the remaining part of the code. JIT-compilation makes the execution slightly slower than that of the AOT-compiled Cython. Parallelization was done using the Scikit-learn library with the IPP backend.

4.4.6 IPP and Loky (CPU)

This section shows the serial and parallel processing performance of loky and IPP implementations running on Seq-X node CPUs. Loky (loky Python library) is limited to a single compute node, while IPP (IPP Python library) has no such limitation, and both are used with the Scikit-learn library. In addition, IPP executions were evaluated for NUMA optimization that evenly distribute the processes among the processor cores, using the *cpu_bind* option with the attribute *distribution=block:cyclic* in the Slurm script, as described in the Section 2.16, and referred below as the Bind option.

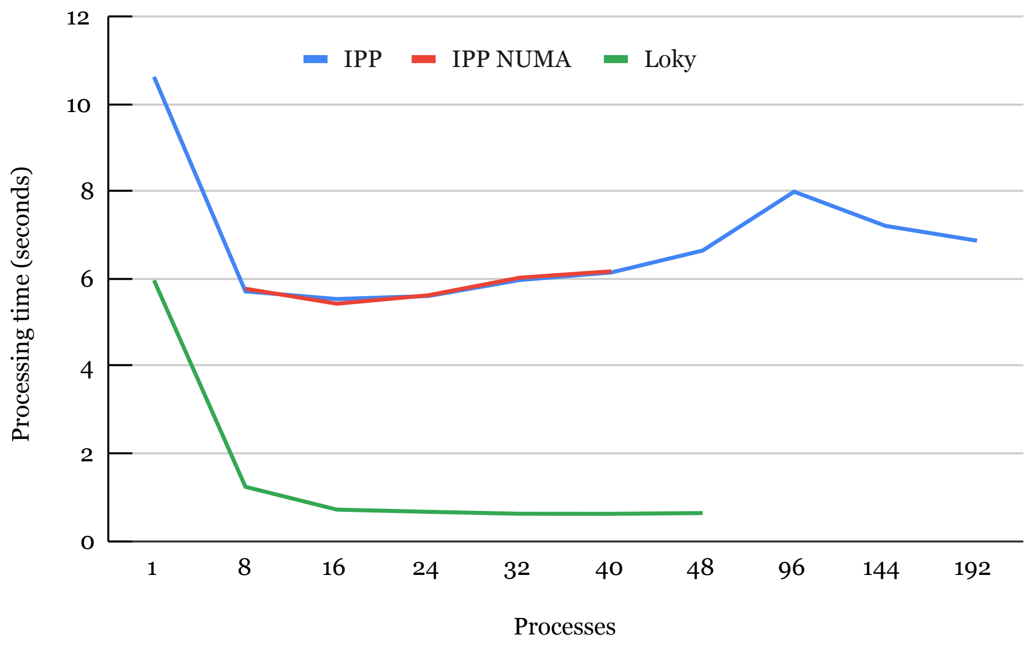
Table 4.9, Figure 4.19, Figure 4.20, and Figure 4.21 show the Random Forest test case processing times for the different implementations and number of processes (loky is restricted to the 48 cores of a single node) executed on Seq-X nodes. The table shows the processing times for the serial and parallel versions with 1 node (up to 48 processes), 2 nodes (96 processes), 3 nodes (144 processes) and 4 nodes (192 processes). In the table, the best times are shown in red, and the sequential Python implementation used as a reference is shown in blue. The shortest time was 0.62 s for the loky implementation with 40 processes, while the worst time was 10.6 s for the 1-process parallel IPP. The IPP implementation optimized for NUMA did not improve the performance in comparison to its non-optimized version. Even limited to a single computing node, loky implementation achieved a far better performance than the IPP one.

Table 4.9 - Performance (Random Forest test case, Seq-X nodes) of the IPP, IPP NUMA and locky implementations as a function of the number of processes: processing times, speedups, and parallel efficiencies. The best values for serial or for each number of processes are shown in **red**. The serial code execution time was taken as a reference for the speedup calculation, shown in **blue**.

Implemen-		Number of processes									
tation	**Seq.**	**1**	**8**	**16**	**24**	**32**	**40**	**48**	**96**	**144**	**192**
					Processing times [s]						
IPP	**8.99**	10.62	5.71	5.53	5.60	5.97	6.14	6.64	**7.99**	**7.21**	**6.87**
IPP NUMA			5.77	5.43	5.62	6.02	6.17				
Loky		**5.96**	**1.24**	**0.72**	**0.67**	**0.62**	**0.62**	**0.64**			
					Speedup						
IPP	**1.00**	0.85	1.57	1.62	1.60	1.51	1.46	1.35	**1.12**	**1.25**	**1.31**
IPP NUMA			1.56	1.66	1.60	1.49	1.46				
Loky		**1.51**	**7.26**	**12.56**	**13.51**	**14.47**	**14.50**	**14.12**			
					Parallel efficiency						
IPP	**1.00**	0.85	0.20	0.10	0.07	0.05	0.04	0.03	**0.01**	**0.01**	**0.01**
IPP NUMA			0.19	0.10	0.07	0.05	0.04				
Loky		**1.51**	**0.91**	**0.79**	**0.56**	**0.45**	**0.36**	**0.29**			

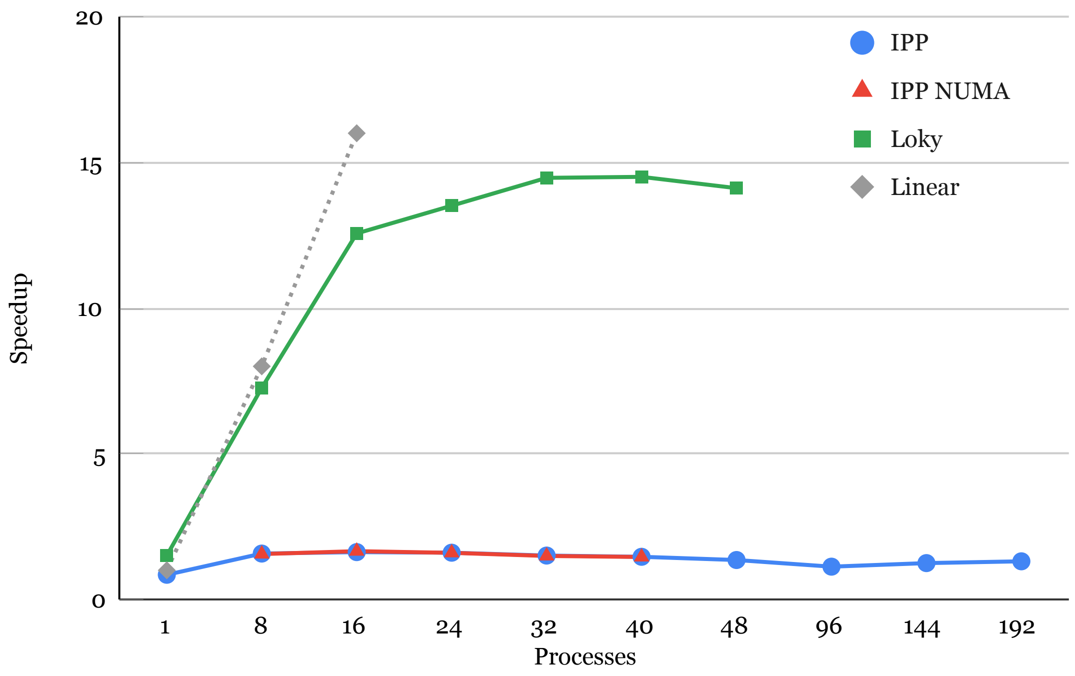
Source: Author's production.

Figure 4.19 - Processing times (seconds) (Random Forest test case, Seq-X nodes) of implementations as a function of the number of processes.



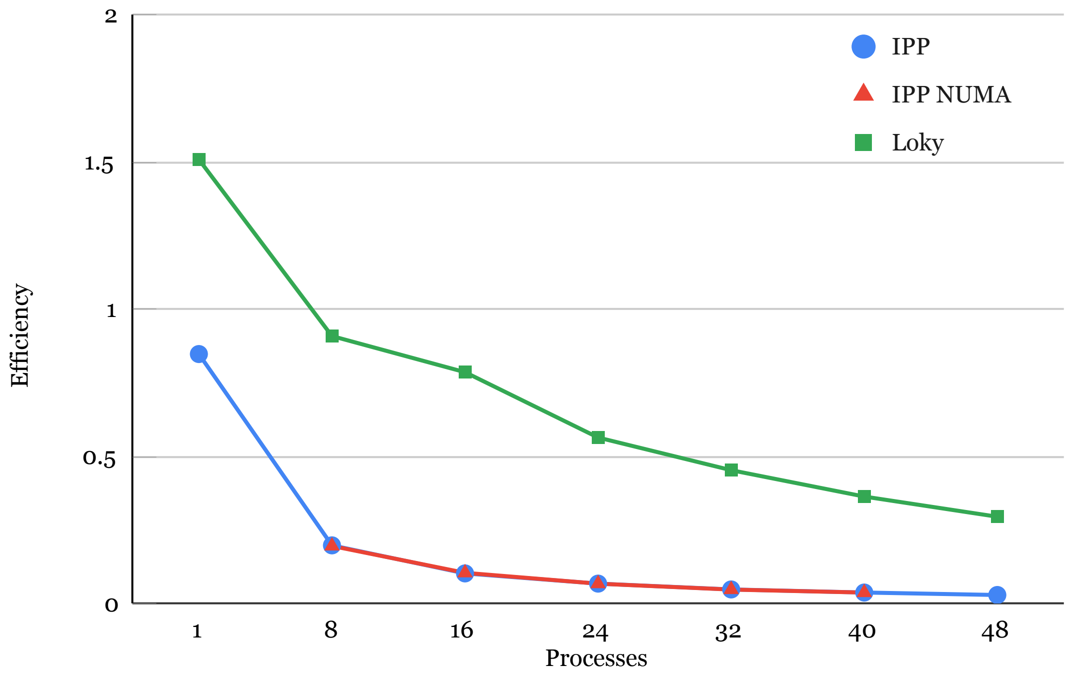
Source: Author's production.

Figure 4.20 - Speedups (Random Forest test case, Seq-X nodes) of 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.

Figure 4.21 - Parallel efficiencies (Random Forest test case, Seq-X nodes) of implementations as a function of number of processes.



Source: Author’s production.

5 SOME PROFILING RESULTS

This chapter shows profiling results only for the stencil and the FFT test cases for both the F90 and F2PY implementations. For convenience, two profilers available in the SDumont programming environment were chosen, the Intel APS and the Python cProfile, both shown in the following sections. Obviously, there are many other profilers that can be applied to the different implementations shown in this work. For instance, Intel APS can be used for Cython, or Numba-GPU may use the Nvidia profiler. However, considering the SDumont programming environment, profiling schemes for some Python implementations may require a careful planning. One example is the batch job submission provided by Slurm that may limit profiling to the post-processing data of the job. Another issue is the code optimization cycle (performing profiling, code optimization and profiling again) may be very limited due to the waiting time in the SDumont Slurm submission queues. Therefore, if possible, such experiments can be performed in a laptop, in order to speed-up eventual optimizations of the code.

All results shown in this chapter are for the stencil and FFT F90 and F2PY implementations generated using the Intel F90 compiler and Intel Python to be compatible to the employed Intel profiling tools.

Performance optimization of serial or parallel software is based on an analysis of execution times and how these times are divided among the several parts or subroutines/functions of the software, or even to identify compute-intensive parts inside specific programs or subroutines/functions. This is usually referred to as performing the timing and profiling of the software. Serial and parallel performance results shown in Chapter 4 were based on timing information, while this chapter shows mainly profiling results.

The profiling of a program comprehends to instrument and run it in order to obtain information about its execution, such as execution time, breakdown of the execution times of the program components (functions, routines, etc.), memory usage, graph of the program call tree, optimizations performed on the source code by the compiler, etc.

Profiling can be performed by the compiler profiling options, which require to recompile the code, by accessing CPU or GPU hardware counters, by adding calls to operational system routines in the original code or by linking the program to a profiling library or program. The resulting profile information allows to analyse performance bottlenecks in the code and thus to optimize its processing performance in terms of time or memory. Usually, it is possible to refer to these profiling approaches as being performed by profilers that may include tools for generating statistics or visualizing profile data.

Except for the use of hardware counters, profiling increases the execution time, since it requires additional calls to time and profiling routines. The original code is then said to be instrumented, and the resulting profiling may be slightly different from the (unknown) real one. Another point is to perform the profiling of the code for problem size and input data similar to those employed for the intended use of the program. In this work, a few profiling tools or profilers were used according to the implementation, all of them available on the SDumont computing environment. It is not intended to provide an overview of all existing profilers, but just to present examples of profiling for some of the implementations.

5.1 The Intel Application Performance Snapshot (APS)

The Intel Application Performance Snapshot (APS) was the profiler employed in this work for all F90 implementations. It is a lightweight profiling tool that supports programs parallelized with using MPI and/or OpenMP libraries (Figure 5.1). The APS requires compilation using the Intel suite of compilers and tools, which include the APS tool. Some specific options must be specified in the compilation, like the option to use the PAPI library, while the program execution is triggered by the APS command with options to specify the output wanted by the user. During the execution of the program, APS stores profile data in disk, being such data available to be read as plain text or by an HTML browser. Table 5.1 shows the performance metrics provided by APS.

Intel provides a wide range of tools, including a more comprehensive set of profilers. For instance, the APS full set includes the Intel VTune Amplifier, to provide more detailed profile data or the Intel Advisor which provides optimization hints.

Even called as a lightweight profiler, Intel APS provides much performance data, as can be seen in Section 6.3 for the stencil and FFT test case F90 implementations. Besides usual timing and profiling data, cache stalls are expressed in percentage of cycles [%c], and (pipeline) memory stalls, in percentage of pipeline slots [%ps]. The *Bound* feature of the profiler indicates if the program is memory-bound or MPI-bound, i.e., limited by memory performance or by the MPI performance, and thus could have its performance improved by optimizing memory access or by the MPI optimization,

Table 5.1 - Intel APS summary of performance metrics.

Metric	Description
Elapsed Time	Execution time of specified application in seconds.
SP GFLOPS	Number of single precision giga-floating point operations calculated per second.
Cycles per Instruction Retired (CPI)	The amount of time each executed instruction took measured by cycles
MPI Time	Average time per process spent in MPI calls
MPI Imbalance	CPU time spent by ranks spinning in waits on communication operations
OpenMP Imbalance	Percentage of elapsed time that your application wastes at OpenMP synchronization barriers because of load imbalance
CPU Utilization	Estimate of the utilization of all logical CPU cores on the system by your application
Memory Stalls	Indicates how memory subsystem issues affect application performance
FPU Utilization	The effective FPU usage while the application was running
I/O Operations	The time spent by the application while reading data from the disk or writing data to the disk
Memory Footprint	Average per-rank and per-node consumption of both virtual and resident memory

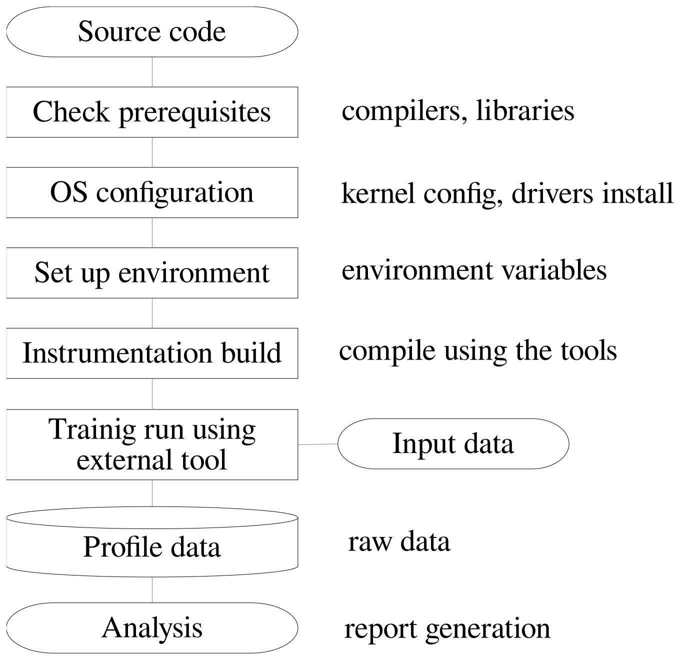
Source: Adapted from Intel APS (2022).

respectively. These hints are based on a set of specific thresholds, as follows: MPI% (10%), IPC (instructions per cycle per core, 1), 20% for Cache%, DRAM%, and Mem Stalls%, and Vectorization% (70%). Poor MPI parallel performance can be due to wait times within the MPI library, sending and receiving messages that are out of sync, imbalance between processes, misconfigurations, etc.

The Intel APS profiler outputs some normalized values, calculated in function of their nominal maximum values. In the case of SDumont B710 nodes, nominal maximum values are: IPC = 4 (according to Intel APS); DP = 19.2 GFlops/sec per core (2.4 GHz $\times$ 8 DP/cycle considering vectorization with E5-2695v2 Ivy Bridge AVX-256 instructions); and DRAM = 59.7 GB/s (for DDR3-1866).

In general, increasing the number of MPI processes, there is a tendency of the program shifting from memory-bound to MPI-bound. This can be expected since the higher the number of processes, the lower the memory demand per process. In addition, increasing the number of processes, the MPI% time also increases, but DP instruction rate typically decreases, since MPI communication and synchronization times increase. In other words, program granularity worsen, since there is less calculation and more communication per process. DP GFlop/sec value per core increases with the number of processes since each process tends to be less constrained by the memory (less memory stalls in the pipelines, less cache misses, etc.). IPC rate tends to be relatively

Figure 5.1 - Intel APS profiling.



Source: Adapted from Intel APS (2022).

for the serial program, lower for a small number of MPI processes, but tends to increase for higher number of processes, since there is a higher number of instructions required by calls to the MPI library.

5.2 The Intel Python Profiler

Python standard library provides the *cProfile*. This Python tool profiles only the part of the Python code, without internally profiling the eventual C/F90 compiled code that may be part of the Python program. It yields the same profile data provided by the standard *gprof* profiler. The Python environment also provides the `timeit` tool to measure the elapsed time based on the system wall clock. It was also employed in this work, but it is worth to note that the profiling performed by the *cProfile* is restricted to the top layer of the Python program, i.e., the Python interpreter.

5.3 Profiling results

Profiling results for the stencil and the FFT test cases for both the F90 and F2PY serial and parallel implementations follow. Intel APS was used for the F90 implementations, and the Python cProfile library was used for the F2PY ones.

5.3.1 Stencil test case profiling for the F90 code

The Intel APS profiler was used to analyze the F90 serial and parallel (MPI) implementations of the Stencil test case (4800 × 4800 point grid, 500 iterations, and 1 energy unit). Code is compiled by the Intel F90 compiler with the APS compilation options, then is executed by using the APS command with the specific options for the program execution, and then APS executed again to generate the profiling reports. In particular, JupyterLab was used in all these steps to allow easy reproducibility and documentation. The stencil F90 code was executed in a SDumont B710 node, being compiled with the Intel Parallel Studio XE 2020 (PSXE). The related PSXE module includes the Intel MPI library.

Table 5.2 shows selected profiling results for the Stencil test case generated by the Intel APS (speedup and parallel efficiency were calculated apart). The serial execution time used as a reference for the speedup calculations is highlighted in **blue**. **Orange** values represent manually calculated values. Some values in the table are void, since they are not applicable, or for some reason, the profiler was not able to calculate them. This table shows that above 16 MPI processes, the *Bound* hint (highlighted in **green**) shifts from memory-bound to MPI-bound. It is also worth to note the amount of time demanded by the MPI_Init function, reaching 35% of the elapsed time (or 48% of the MPI time) for 81 processes.

It is important to note that times, speedups and parallel efficiencies shown in Table 5.2 are different to the corresponding performance results shown in Chapter 4, since the results shown in this table were obtained using the Intel F90 compiler, not the GNU F90 compiler used in that chapter. However, performance results in the table are worst mainly due to the profiler overhead, which tends to increase with the number of MPI processes. In the same table, performance results that have room to be improved are indicated by APS, and are highlighted in **red** in this table.

Figure 5.2 shows profiling results obtained for the Stencil test case F90 implementation as a function of MPI processes, generated by the Intel APS profiler, according to the definitions shown in the Section 5.1. This figure shows a decrease of the normalized DP GFlop/sec (total) value with the increase of the number of MPI processes, since the nominal DP GFlop/sec total value tends to increase much more. The profiling information can be further employed to optimize the code or to investigate some kind of performance anomaly. For instance, in the same figure, results for 9 MPI processes presented the highest percentage of memory stalls and the lowest IPC rate.

Table 5.2 - APS-generated profiling results (Stencil test case, B710 nodes) for the F90 implementation as a function of the number of MPI processes. Manually calculated values are highlighted in orange. The serial execution time used as a reference for the speedup calculations is highlighted in blue. Performance results that have room to be improved are highlighted in **red**. *Bound* hint is highlighted in green.

		Number of MPI processes							
Parameter	**Serial**	**1**	**4**	**9**	**16**	**36**	**49**	**64**	**81**
Elapsed Time [s]	22.6	23.3	9.1	7.9	7.5	5.0	4.8	5.4	5.7
NonMPI Time [s]	-	**22.8**	**8.1**	**6.7**	**5.5**	**2.4**	**2.0**	**1.8**	**1.5**
Speedup	-	**1.0**	**2.5**	**2.9**	**3.0**	**4.5**	**4.7**	**4.2**	**4.0**
Efficiency	-	**1.0**	**0.6**	**0.3**	**0.2**	**0.1**	**0.1**	**0.1**	**0.0**
MPI Time [s]	-	0.5	1.0	1.2	2.0	2.6	2.8	3.6	4.2
MPI Time [%]	-	2.1	**10.9**	**15.1**	**27.5**	**52.5**	**59.4**	**68.2**	**74.4**
MPI_Init [s]	-	1.0	0.8	1.0	1.1	1.5	1.4	1.9	2.0
MPI_Wait [s]	-	-	0.2	0.1	0.9	0.9	1.0	0.8	1.2
MPI_Bcast [s]	-	-	0.0	0.0	0.0	0.2	0.3	0.8	0.8
MPI Imbalance [s]	-	-	0.1	0.0	0.7	0.9	1.0	1.2	1.6
DP [GFlops]	3.8	3.6	9.6	12.4	12.0	17.2	16.4	15.4	13.6
IPC Rate	1.5	1.3	**1.0**	**0.6**	**0.9**	1.3	1.5	1.5	1.7
Bound	**mem**	**mem**	**mem**	**mem**	**mem**	**MPI**	**MPI**	**MPI**	**MPI**
Cache Stalls [%c]	13.0	14.4	**23.8**	**32.2**	**26.9**	16.2	16.3	12.6	10.2
DRAM Stalls [%c]	13.3	14.3	**29.2**	**33.2**	**27.7**	**24.4**	14.0	10.6	7.1
DRAM [GB/s]	-	11.9	26.4	-	28.5	20.0	12.1	12.5	8.0
Mem Stalls [%ps]	**29.3**	**31.9**	**44.7**	**67.3**	**52.9**	**36.2**	**26.1**	**21.7**	14.0
Vectorization [%]	100.0	100.0	100.0	100.0	100.0	99.9	98.5	99.8	98.0

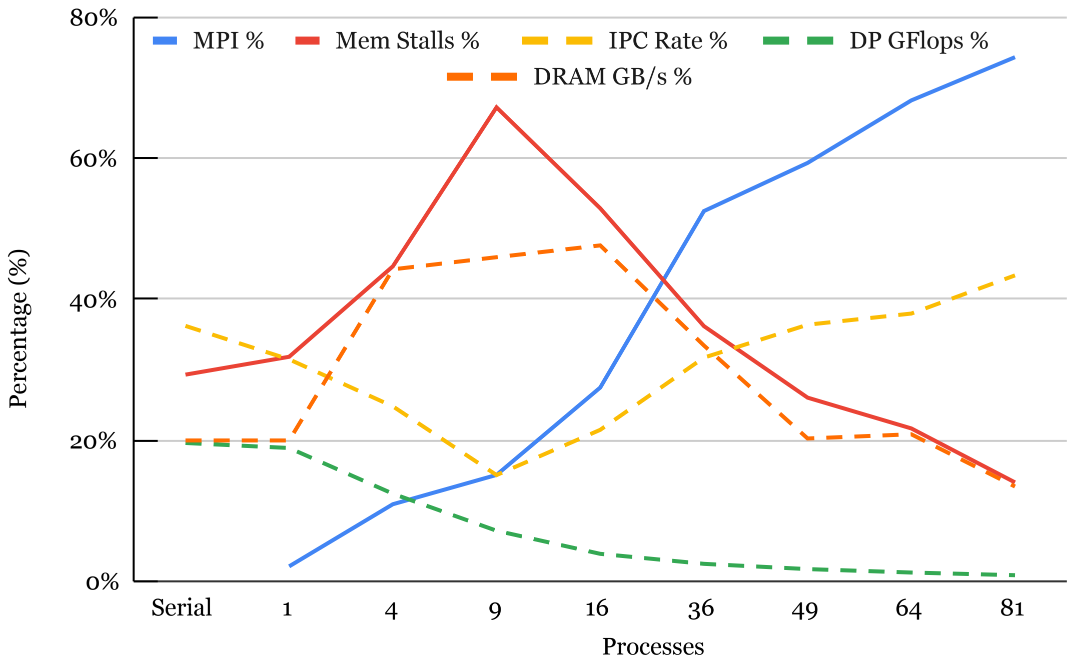
Source: Author's production.

Figure 5.3 shows the processing times for the F90 Stencil test case as a function of the number of MPI processes. As expected, the increase of the number of MPI processes decreases the elapsed time, while increasing the MPI time and decreasing the difference between these times.

5.3.2 FFT test case profiling for the F90 code

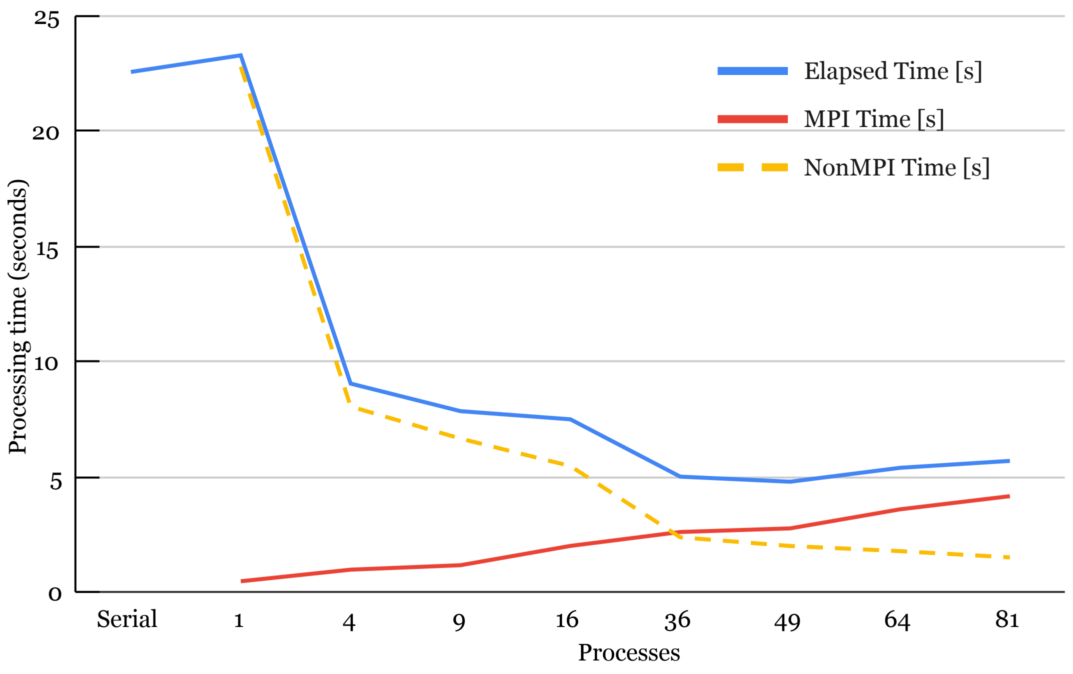
Similarly to the previous section, the Intel APS profiler was used to analyze the F90 serial and parallel (MPI) implementations of the FFT test case (3D array of complex numbers with dimension $576 \times 576 \times 576$). Code was compiled by the Intel F90 compiler with the APS compilation options, then is executed by using the APS command with the specific options for the program execution, and then APS executed again to generate the profiling reports. In particular, JupyterLab was used in all these steps to allow easy reproducibility and documentation. The FFT F90 code was executed in a SDumont B710 node, being compiled with the Intel Parallel

Figure 5.2 - Profiling values (Stencil test case, B710 nodes) for the F90 implementation as a function of the number of MPI processes. Except for MPI% and Mem Stalls% (continuous lines), percentage values (dotted lines) result from normalization taking the corresponding maximum values of each variable.



Source: Author's production.

Figure 5.3 - Processing times (seconds) (Stencil test case, B710 nodes) for the F90 implementation as a function of the number of MPI processes.



Source: Author's production.

Studio XE 2020 (PSXE). The related PSXE module includes the Intel MPI library.

Table 5.3 shows the profiling results selected for the FFT F90 test case generated by the Intel APS (speedup and parallel efficiency were calculated apart). The serial execution time used as a reference for the speedup calculations is highlighted in blue. Some values in the table are void, since they are not applicable, or for some reason, the profiler was not able to calculate them. This table shows that above 4 MPI processes, the *Bound* hint shifts from memory-bound to MPI-bound. It is also worth to note the amount of time demanded by the MPI_Init function, reaching 47% of the elapsed time (or 63% of the MPI time) for 96 processes.

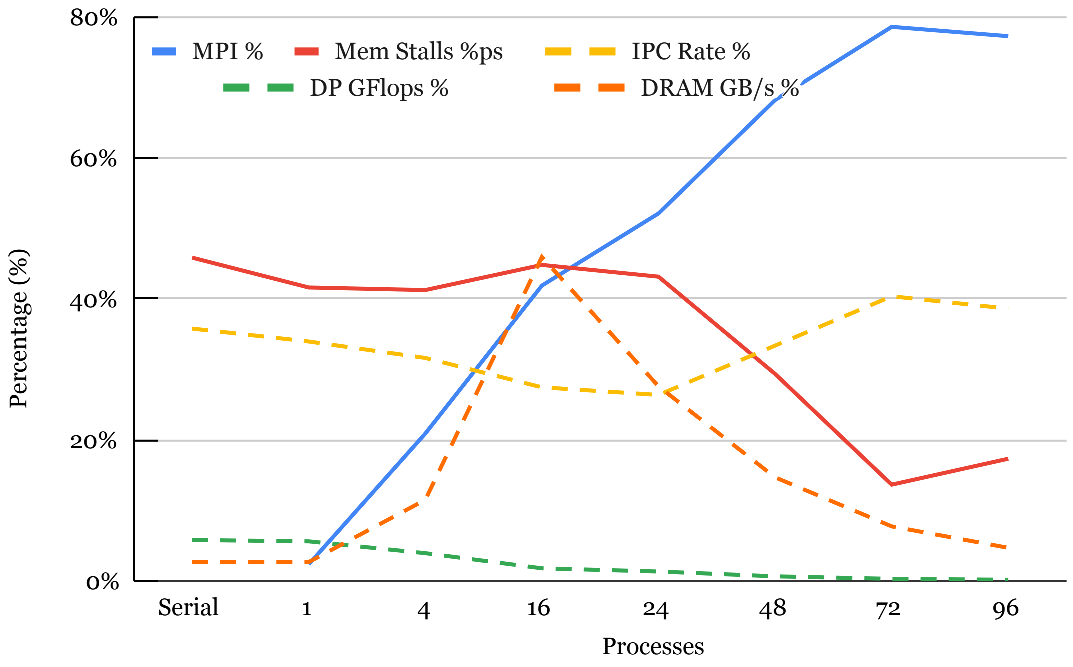
Table 5.3 - APS-generated profiling results (FFT test case, B710 nodes) for the F90 implementation as a function of the number of MPI processes. Manually calculated values are highlighted in orange. The serial execution time used as a reference for the speedup calculations is highlighted in blue. Performance results that have room to be improved are highlighted in red. *Bound* hint is highlighted in green.

		Number of MPI processes						
Parameter	**Serial**	**1**	**4**	**16**	**24**	**48**	**72**	**96**
Elapsed Time [s]	20.3	21.1	7.4	4.0	4.0	3.8	5.1	5.5
NonMPI Time [s]	-	**20.6**	**5.8**	**2.3**	**1.9**	**1.2**	**1.3**	**1.3**
Speedup	-	**1.0**	**2.7**	**5.1**	**5.1**	**5.4**	**4.0**	**3.7**
Efficiency	-	**1.0**	**0.7**	**0.3**	**0.2**	**0.1**	**0.1**	**0.0**
MPI Time [s]	-	0.5	1.5	1.7	2.1	2.5	3.7	4.1
MPI Time [%]	-	2.4	**20.9**	**41.9**	**52.2**	**68.3**	**78.7**	**77.3**
MPI_Init [s]	-	0.5	0.7	1.0	1.5	1.8	2.2	2.6
MPI_Sendrecv [s]	-	-	0.8	0.6	0.4	0.3	0.5	0.3
MPI Imbalance [s]	-	-	0.1	0.3	0.2	0.4	1.2	1.1
DP [GFlops]	1.1	1.1	3.0	5.6	6.3	6.2	4.5	3.6
IPC Rate	1.4	1.4	1.3	1.1	1.1	1.3	1.6	1.5
Bound	mem	mem	mem	MPI	MPI	MPI	MPI	MPI
Cache Stalls [%c]	9.1	8.8	19.2	17.9	18.0	13.7	9.3	10.6
DRAM Stalls [%c]	**32.2**	**29.2**	16.0	**21.0**	**21.5**	12.3	5.7	4.9
DRAM [GB/s]	-	1.6	6.9	15.5	16.5	8.8	4.6	2.8
Mem Stalls [%ps]	**45.9**	**41.7**	**41.3**	**44.9**	**43.2**	**29.4**	13.7	17.4
Vectorization [%]	**5.3**	**5.5**	**5.3**	**5.4**	**7.8**	**5.0**	**31.0**	**18.4**

Source: Author's production.

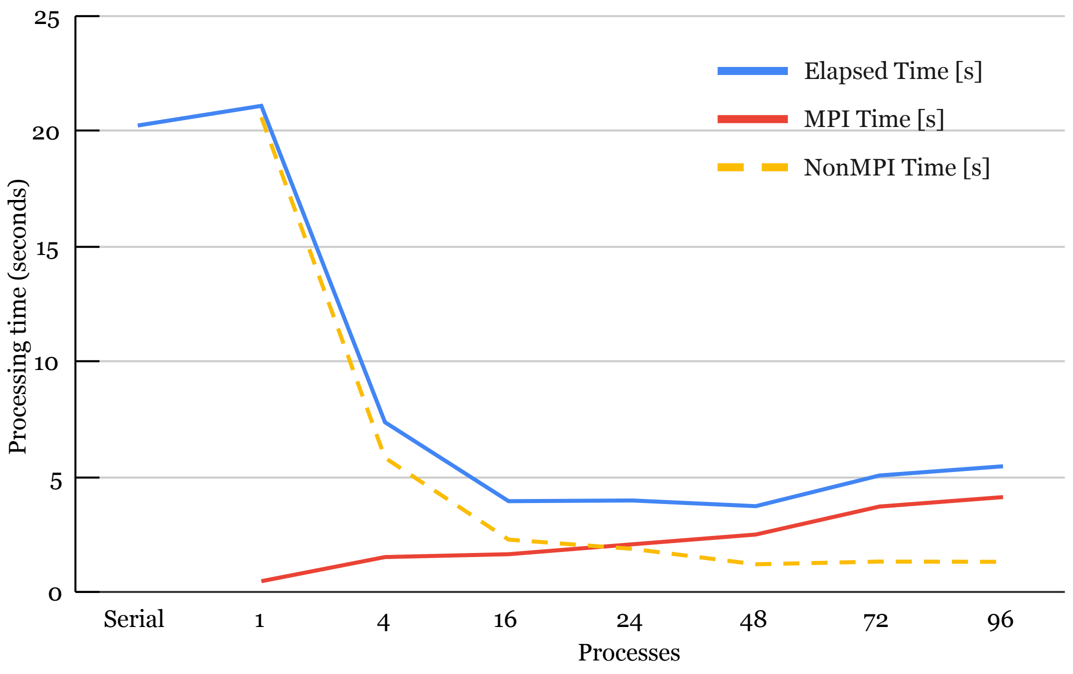
It is important to note that times, speedups and parallel efficiencies shown in Table 5.3 are different to the corresponding performance results shown in Chapter 4, since the results shown in this table were obtained using the Intel F90 compiler, not the GNU F90 compiler used in that chapter. However, performance results in Table 5.3 are

Figure 5.4 - Profiling values (FFT test case, B710 nodes) for the F90 implementation as a function of the number of MPI processes. Except for MPI% and Mem Stalls%ps (continuous lines), percentage values (dotted lines) result from normalization taking the corresponding maximum values of each variable.



Source: Author's production.

Figure 5.5 - Processing times (seconds) (FFT test case, B710 nodes) for the F90 implementation as a function of the number of MPI processes.



Source: Author's production.

worst mainly due to the profiler overhead, which tends to increase with the number of MPI processes. In the same table, performance results that have room to be improved are indicated by APS, and are highlighted in **red** in this table. Figure 5.4 shows profiling results obtained for the FFT test case F90 implementation as a function of MPI processes, generated by the Intel APS profiler, according to the definitions shown in the Section 5.1. This figure shows a decrease of the normalized DP GFlop/sec (total) value with the increase of the number of MPI processes, since the nominal DP GFlop/sec total value tends to increase much more. The profiling information can be further employed to optimize the code or to investigate some kind of performance anomaly. For instance, in the same figure, results for 16 MPI processes presented the highest percentage of memory stalls, while the lowest IPC rate occurred for 24 processes.

Figure 5.5 shows the processing times for the F90 FFT test case as a function of the number of MPI processes. As expected, the increase of the number of MPI processes decreases the elapsed time, while increasing the MPI time and decreasing the difference between these times.

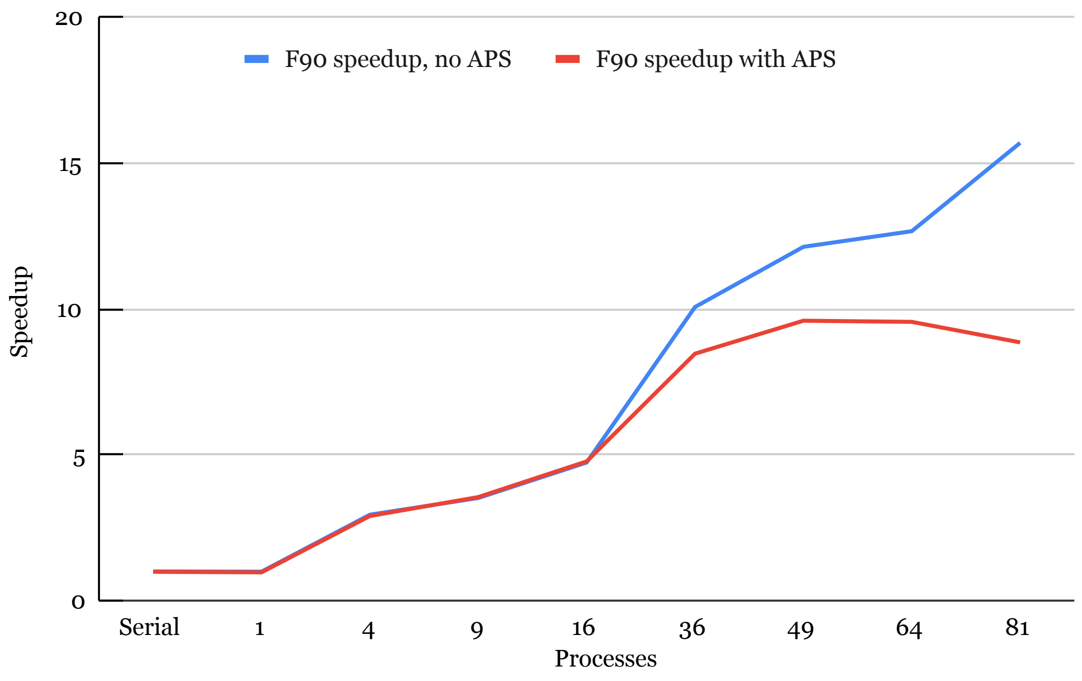
5.4 Overhead due to the APS Profiler in the F90 codes

Any profiler incorporate in the original code calls for functions or routines that perform timing and profiling. As a consequence, the resulting instrumented code will have an overhead of execution time, the profiling code-instrumentation overhead. This section shows the instrumentation overhead of the Intel APS profiler for both the stencil and FFT serial and parallel implementations in F90 with and without the profiler. These implementations were compiled with the Intel F90 compiler using the optimization flag -O3, and execution time values are given by the average of 3 runs in the SDumont B710 nodes.

Figure 5.6 compares the speedups for the F90 Stencil test case with and without the Intel APS profiler as a function of the number of MPI processes. As already commented, the profiler overhead can much affect the speedup of the MPI program, tending to increase with the number of MPI processes. In this case, the profiler overhead peaks at 81 processes, reducing the original speedup from 15.7 to 8.8 (44% reduction).

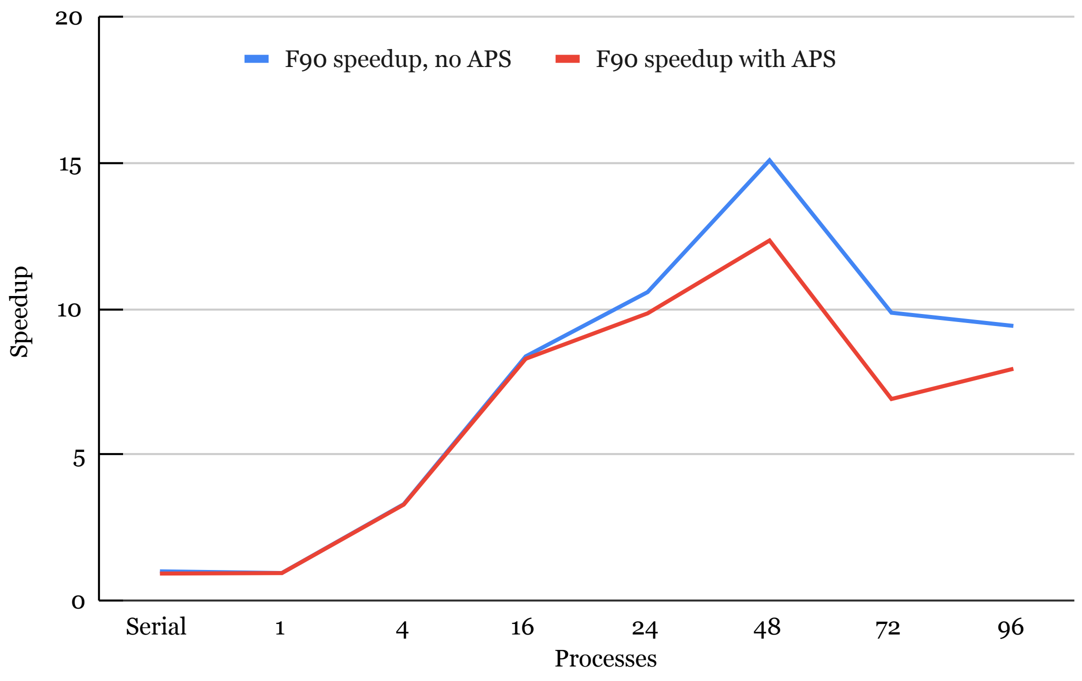
Figure 5.7 shows the comparison of speedups with Intel APS (in red) and without Intel APS (in blue), as a function of number of MPI processes, including serial version, for the F90 implementation, running on B710 nodes, and using Intel tools.

Figure 5.6 - Comparison of the speedups (Stencil test case, B710 nodes) for the F90 implementation with and without the Intel APS profiler as a function of the number of MPI processes.



Source: Author's production.

Figure 5.7 - Speedups (FFT test case, B710 nodes) as a function of the number of MPI processes, including the serial version, for the F90 implementations, built with and without the Intel APS.



Source: Author's production.

The comparison shows the effect of using the Intel APS library on the results of the F90 implementation. As the number of MPI processes increases, the influence of Intel APS increases, and in 72 processes the speedup was reduced from 10 to 7 (30% reduction), making the analysis of profiling results non-trivial, needing to consider deviations, and possibly using other profiling tools when there is a desire to get results close to the result of the implementation that does not use profiling.

5.5 The overhead due to the use of Python in the F2PY codes

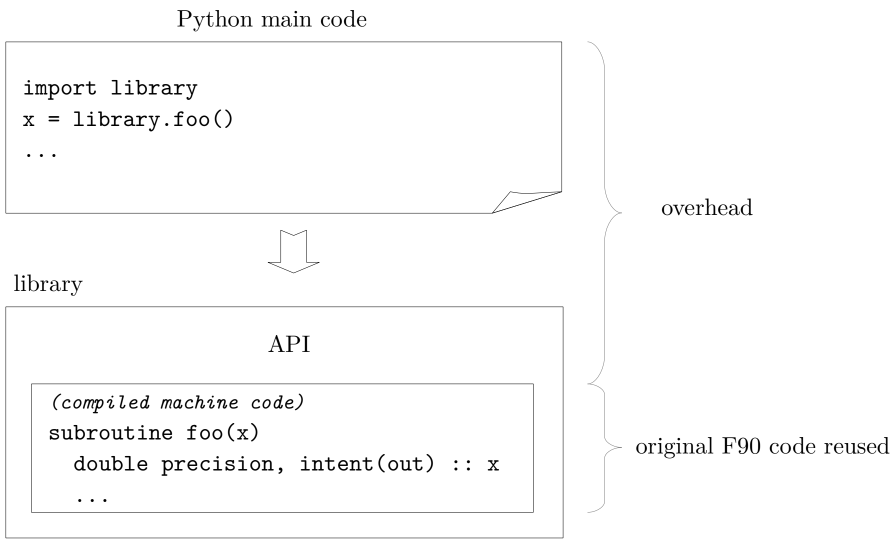
A question that arises is how much the performance is affected when using a Python program in comparison to the straightforward use of a C/F90 compiled program. The overhead due to the use of Python is expected, since a Python program may reuse a C/F90 compiled program or library, for instance. Obviously, the use of Python adds an extra level of programming due to the Python commands. On the contrary, such overhead may be acceptable, since Python programming is typically easier and allows a modular approach.

In this work, the Python overhead was evaluated for the F2PY implementations of both the stencil and FFT test cases. This overhead is depicted for F2PY in Figure 5.8. F2PY is a feature of the NumPy library that reuses one or more original F90 subroutines, which are then incorporated as functions from a standard Python library. However, each F90 routine must be called as a Python function and thus a F2PY wrapper is required to generate the APIs that maps Python function calls to the corresponding F90 subroutines embedded in the related library. The execution of the standard Python code is done in an interpreted way by the Python virtual machine, while the execution of the function of F2PY-generated Python library is done by the compiled F90 native code.

These Python overhead measures were performed using different execution time data, generated by calls to the system wall time, use of the Python cProfile, and calls to F90 CPU_TIME routine in the original code. Each measure of the Python overhead is assumed as given by the difference between the F2PY and F90 implementations.

Even considering only timing information, values of execution time may be different according to the employed tool. Table 5.4 and Table 5.5 show, respectively, the execution times of the F90 and F2PY implementations for the stencil and FFT test cases in function of the number of MPI processes. These tables compare execution times obtained by different tools. The Intel Python cProfile was used only for timing, while Intel APS profiler was not employed. Each profiling value is the average of 3

Figure 5.8 - Python overhead when using F2PY.



```
import library
x = library.foo()
...
```

```
(compiled machine code)
subroutine foo(x)
  double precision, intent(out) :: x
  ...
```

Source: Author's production.

runs of each implementation. The shortest execution times for each number of MPI processes are highlighted in red. The different timing tools employed here follow. Except for the last two cProfile tools, the desired elapsed time is yielded by the difference between two successive wall time values.

Table 5.4 - Processing times (seconds) (Stencil test case, B710 nodes) for the F90 (without the Intel APS profiler) and F2PY (with timing information of the Intel cProfiler) implementations as a function of the number of MPI processes.

Implemen-		**Number of MPI processes**								
tation	**Serial**	**1**	**4**	**9**	**16**	**36**	**49**	**64**	**81**
					F90					
OS time command	21.94	22.56	7.98	6.98	5.63	3.81	3.49	4.08	3.85
F90 wall time call	**21.88**	22.04	7.43	6.22	4.62	2.17	**1.80**	1.73	**1.39**
					F2PY					
OS time command	22.48	21.04	6.83	5.28	4.04	3.44	4.19	4.50	4.70
Python wall time call	21.95	20.43	6.24	4.61	3.33	2.67	3.48	3.74	3.84
F90 wall time call	21.94	**20.02**	**5.81**	**4.02**	**2.35**	**1.41**	1.83	**1.60**	1.67
cProfile tottime	21.95	20.43	6.24	4.61	3.33	2.67	3.48	3.74	3.84
cProfile cumtime	22.33	20.88	6.67	5.08	3.84	3.24	3.82	4.30	4.49

Source: Author's production.

- **OS time command**: uses only the wall time value (CPU time, etc. was

Table 5.5 - Processing times (seconds) (FFT test case, B710 nodes) for the F90 (without the Intel APS profiler) and F2PY (with timing information of the Intel cProfiler) implementations as a function of the number of MPI processes.

Implementation	Serial	Number of MPI processes 1	4	16	24	48	72	96
F90								
OS time command	18.76	20.13	6.42	3.22	3.55	3.22	4.22	4.27
F90 wall time call	**18.57**	**19.70**	**5.62**	**2.22**	**1.76**	**1.23**	**1.88**	**1.97**
F2PY								
OS time command	20.36	20.85	6.82	4.08	4.28	4.12	4.76	5.42
Python wall time call	19.77	20.29	6.23	3.36	3.48	3.37	3.91	4.64
F90 wall time call	19.76	19.88	5.78	2.35	1.99	1.64	1.93	2.03
cProfile tottime	19.78	20.29	6.23	3.36	3.47	3.39	3.90	4.64
cProfile cumtime	20.20	20.70	6.66	3.88	4.06	3.90	4.52	5.21

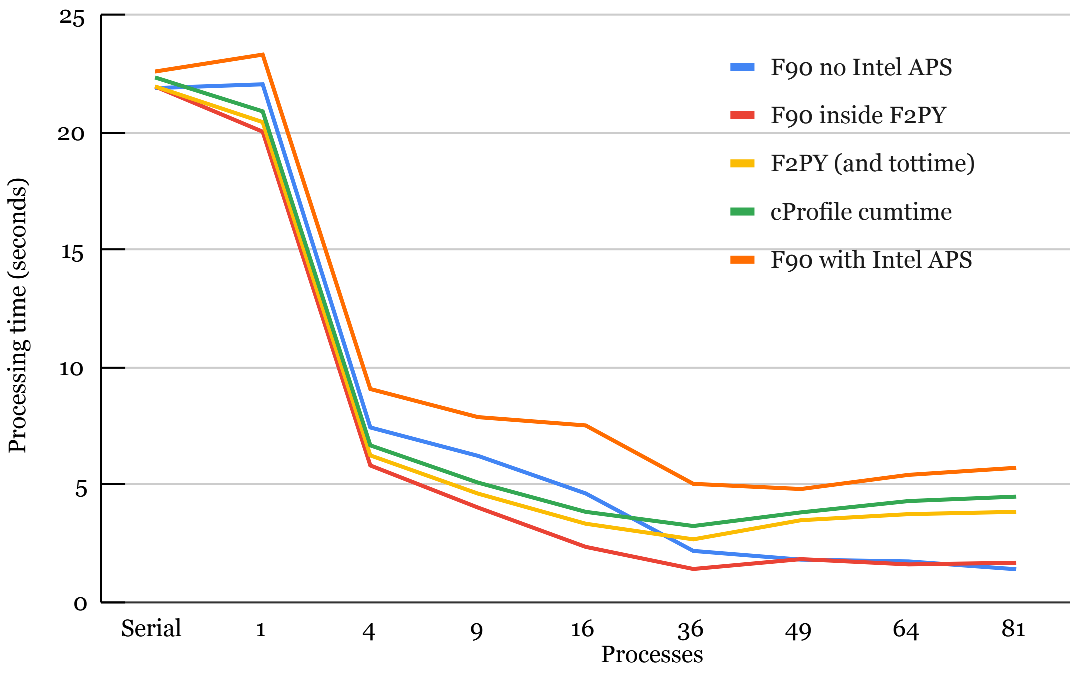
Source: Author's production.

not used);

- **F90 wall time call (F90 code alone)**: calls embedded in the F90 code;
- **Python wall time call**: analogous to the OS time command;
- **F90 wall time call (F90 code wrapped by F2PY)**: calls embedded in the wrapped F90 code;
- **cProfile tottime**: elapsed time calculated using Python library wall time calls to F90 routines wrapped into Python library functions;
- **cProfile cumtime**: idem, but adding time due to the Python overhead.

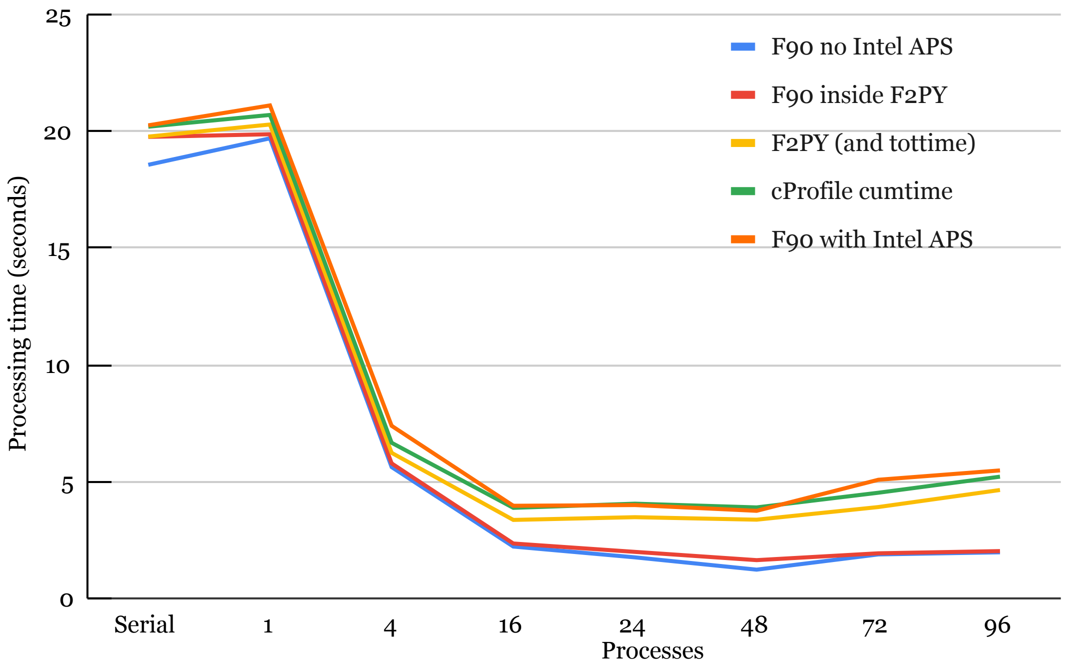
Figure 5.9 shows the processing times (seconds) of the F90 and F2PY implementations of the Stencil test case as a function of the number of MPI processes, using the values already shown in Table 5.4. Figure 5.10 shows the processing times (seconds) of the F90 and F2PY implementations of the FFT test case as a function of the number of MPI processes, using the values already shown in Table 5.5.

Figure 5.9 - Processing times (seconds) (Stencil test case, B710 nodes) as a function of the number of MPI processes for the F90 and F2PY implementations.



Source: Author's production.

Figure 5.10 - Processing times (seconds) (FFT test case, B710 nodes) as a function of the number of MPI processes for the F90 and F2PY implementations.



Source: Author's production.

6 FINAL REMARKS

The Python environment provides fast prototyping of computer code in a high level of abstraction, including Python and third-part libraries. However, the straightforward use of Python is usually slow, since it is an interpreted language, requiring the use of Python HPC approaches. This work presents the most common of HPC approaches applied to three selected test cases, mostly using MPI, which were also implemented in F90/MPI for the purpose of comparison. Some test cases were implemented in the *Scikit-learn* that uses the IPython Parallel library. All implementations were developed and executed on the LNCC Santos Dumont supercomputer using its available resources. The considered three test cases were:

- Stencil test case: a five-point stencil finite difference method to solve partial differential equations resulting from Poisson equations, applied to a 2D heat transfer problem on a finite surface;
- Fast Fourier Transform (FFT) test case: an algorithm that computes the multidimensional Fourier transform of an 3D array of synthetic data;
- Random Forest test case: a random forest algorithm applied for the classification of asteroid orbits of a NASA dataset.

The serial and parallel implementations in F90 of the test cases were taken as a reference to compare their performance with some serial and parallel implementations of the same algorithms using approaches available in the Python environment of the supercomputer: F2PY, Cython, Numba, Numba-GPU, and the standard Python itself. Except for some implementations, parallel code was generated using the MPI library and executed in one or more nodes of the supercomputer using multicore processors. Processing times, speedups and parallel efficiencies were presented and discussed for these implementations considering a specific problem size for each test cases. Profiling was performed only for the F90 and F2PY implementations of the stencil and the FFT test cases. For convenience, JupyterLab notebooks were used, providing a web-based environment that facilitates the exchange, development and execution of code in different environments, and thus improving code documentation, portability and reproducibility.

This work may represent a small primer for using MPI-based HPC features in the Python programming environment, considering execution in multicore processors, but there is a full set of Python alternatives for execution in GPUs. For instance, this work presented a few implementations in Numba-GPU.

It is important to note that there is not a standard approach for achieving HPC in Python, it depends on the available choices in terms of the programming environment and computer hardware, the programmer knowledge or experience, etc. In general, Python code can be optimized by performing timing and profiling in order to identify compute-intensive parts of the code, and then replacing them by more efficient library routines. Python modular approach facilitates such approach. The following section provides some general recommendations.

6.1 Some general recommendations for HPC Python

This section lists below some general recommendations for using HPC resources in Python. However, some of these recommendations may be specific for the test cases employed in this work. Different algorithms or application problems may require different Python HPC approaches.

- In general, in the case of non pre-existing F90 or C code, Python allows rapid prototyping and wide portability. However, since Python is an interpreted language, it is slow. Consequently, compute-intensive parts like loops may compromise processing performance. Therefore, in order to optimize the Python program, these parts must be replaced by calls to more efficient library routines, or re-written in F90 or Cython. The optimization process can be performed in a progressive way, taking advantage of Python interactive and modular nature.

- When starting to write a new Python program, or when only minor changes are intended in an existing Python program, the easier alternative to aim processing performance is Numba, due to its portability and ability to generate code for both CPU and GPU execution, although Numba supports only a subset of Python. Since Numba is compiled JIT, the code can eventually be optimized at execution time for the employed architecture, and thus ensuring portability.

- In the case of existing F90 code, it can be reused and wrapped as a function/routine into a Python library using F2PY, a resource of the NumPy library. Most of the original F90 code can be left unchanged, including the eventual calls to the MPI library.

- In the case of existing F90 code with a known compute-intensive part(s), it is possible to port most of the original F90 code to the Python language, except for that part(s) that may be wrapped into a function/subroutine using F2PY and called from the Python program. This approach takes more time than the former one, but Python resources allow making the new library available for other programs or users in a user-friendly way.

- When there is not an F90 code, Cython is usually a good choice for obtaining processing performance while having a Python-like syntax and advantages similar to those of Python. Cython supports both Python and Cython languages, allowing to integrate Python code. On the other way, compute-intensive parts of a Python code can be ported to Cython to improve processing performance. Similarly to F2PY, Cython builds a standard Python library that can be called from the Python program.

- When writing a Python program to be executed on a GPU, Numba is the best choice, although major code changes are required to obtain maximum GPU performance. In addition, Numba JIT compilation allows the generation of code optimized for the architecture employed for execution.

- As a general rule, the best approach for HPC is to write a standard Python program from scratch, but using off-the-shelf optimized libraries for known compute-intensive parts. Environments like JupyterLab provide documentation, resulting in a code easy to understand, maintain, and reuse. In a second step, profiling may allow optimizing the remaining not-known compute-intensive parts.

- Another general rule is to search for highly optimized libraries for specific classes of algorithms, that may be tailored for CPU or GPU execution (this is not the case of this work).

6.2 Future work

Future work foresees a more detailed analysis of profiling data in order to improve the MPI parallel performance of the test case implementations. This can be achieved by optimizing the existing Python and F90 codes, or by exploring different, more optimized, libraries. It is also foreseen the development of new implementations of these test cases aimed for execution in GPUs, also available in the Santos Dumont supercomputer.

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. 127

AGIUS, I.-M.; INGUANEZ, F. Re-purposing computers as a cluster for academic institutions. In: IEEE; INTERNATIONAL CONFERENCE ON CONSUMER ELECTRONICS, 9., 2019, Berlin. **Proceedings...** [S.l.]: IEEE, 2019. p. 57–60. DOI 10.1109/ICCE-Berlin47944.2019.8966155. 124

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. 124

BALAJI, P.; GROPP, W.; HOEFLER, T.; THAKUR, R. Advanced MPI programming. **Argonne National Laboratory**, ANL, 2017. Tutorial at SC17, the international conference for high performance computing, networking, storage, and analysis. Available from: <http://www.mcs.anl.gov/~thakur/sc17-mpi-tutorial/>. Access in: Dec 2021. 27, 28, 191

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. 3

BEAZLEY, D. M.; LOMDAHL, P. S. Feeding a large-scale physics application to Python. In: INTERNATIONAL PYTHON CONFERENCE, 6., 1997, San Jose, California. **Proceedings...** 1997. OSTI 658327. Available from: <http://legacy.{P}ython.org/workshops/1997-10/proceedings/beazley.html>. Access in: Dec 2021. 1

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. 3, 9, 116

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: PYTHON IN SCIENCE

CONFERENCE, 9., 2010. **Proceedings...** [S.l.], 2010. v. 1, p. 18 – 24. DOI 10.25080/Majora-92bf1922-003. 122

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. 115, 126

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. 118

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)**, v. 3, n. 4, p. 1–es, 2003. DOI 10.1145/1047568.1047569. 125

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**, v. 2, n. 6, p. 493–507, 2012. 18

BREIMAN, L. Random forests. **Machine Learning**, v. 45, n. 1, p. 5–32, 2001. DOI 10.1023/A:1010933404324. 39, 40

BROWNLEE, J. **Deep learning with Python: develop deep learning models on Theano and TensorFlow using Keras**. [S.l.]: Machine Learning Mastery, 2016. 122

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**, v. 55, n. 2, p. B34–B34, 2002. DOI 10.1115/1.1451171. 25

CHEN, Y.-Y.; YU, S.-t. Constructing a supercomputing framework by using Python for hybrid parallelism and GPU cluster. In: AIAA COMPUTATIONAL FLUID DYNAMICS CONFERENCE, 20., 2011. **Proceedings...** [S.l.], 2011. p. 3220. 24

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]. 125

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, 20

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. 121

DAGUM, L.; MENON, R. OpenMP: an industry standard api for shared-memory programming. **IEEE Computational Science and Engineering**, v. 5, n. 1, p. 46–55, 1998. 3

DALCÍN, L.; PAZ, R.; STORTI, M.; D'ELÍA, J. MPI for Python: performance improvements and MPI-2 extensions. **Journal of Parallel and Distributed Computing**, v. 68, n. 5, p. 655–662, 2008. ISSN 0743-7315. DOI 10.1016/j.jpdc.2007.09.005. 2, 7, 116

DANIEL, T. R.; MIRCEA, S. AES on GPU using CUDA. In: EUROPEAN CONFERENCE FOR THE APPLIED MATHEMATICS & INFORMATICS, 2010, Athens, Greece. **Proceedings...** [S.l.]: World Scientific and Engineering Academy and Society Press, 2010. p. 225 – 230. ISBN 978-960-474-260-8. ISSN 1792-7390. 14

DE RAINVILLE, F.-M.; FORTIN, F.-A.; GARDNER, M.-A.; PARIZEAU, M.; GAGNÉ, C. DEAP: a Python framework for evolutionary algorithms. In: ANNUAL CONFERENCE COMPANION ON GENETIC AND EVOLUTIONARY COMPUTATION, 14., 2012. **Proceedings...** [S.l.], 2012. p. 85–92. DOI 10.1145/2330784.2330799. 125

DOBESOVA, Z. Programming language Python for data processing. In: INTERNATIONAL CONFERENCE ON ELECTRICAL AND CONTROL ENGINEERING, 2011. **Proceedings...** [S.l.], 2011. p. 4866–4869. 3

DONGARRA, J. J.; OTTO, S. W.; SNIR, M.; WALKER, D. **An introduction to the MPI standard**. USA: [s.n.], 1995. DOI 10.5555/898812. 7

DRABAS, T.; LEE, D. **Learning PySpark**. [S.l.]: Packt Publishing Ltd, 2017. 126

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. 16

FEY, M.; LENSSEN, J. E. Fast graph representation learning with PyTorch geometric. **CoRR**, 2019. ArXiv: 1903.02428 [cs.LG]. 121

FINK, Z.; LIU, S.; CHOI, J.; DIENER, M.; KALE, L. V. Performance evaluation of Python parallel programming models: Charm4Py and mpi4py. 2021. DOI 10.48550/arXiv.2111.04872. 23

FRIGO, M.; JOHNSON, S. G. FFTW: An adaptive software architecture for the FFT. In: IEE INTERNATIONAL CONFERENCE ON ACOUSTICS, SPEECH AND SIGNAL PROCESSING, 1998. **Proceedings...** [S.l.], 1998. v. 3, p. 1381–1384. 15

GAYER, A. V.; CHERNYSHOVA, Y. S.; SHESHKUS, A. V. Effective real-time augmentation of training dataset for the neural networks learning. In: INTERNATIONAL CONFERENCE ON MACHINE VISION, 2018., 11. **Proceedings...** [S.l.], 2019. v. 11041, p. 394 – 401. DOI 10.1117/12.2522969. 122

GOMERSALL, H. **pyFFTW: Python wrapper around FFTW**. sep. 2021. ascl:2109.009 p. Bibcode: 2021ascl.soft09009G. Astrophysics Source Code Library, record ascl:2109.009. Provided by the SAO/NASA Astrophysics Data System. Available from: <`https://ui.adsabs.harvard.edu/abs/2021ascl.soft09009G`>. Access in: Dec 2021. 15

GONZALEZ, J.; TAYLOR, J.; CASTRO, S.; KERN, J.; KNUDSTRUP, J.; ZAMPIERI, S.; MANNING, A.; BHATNAGAR, S.; DAVIS, L.; GOLAP, K. et al. Python code parallelization, challenges and alternatives. In: ASTRONOMICAL DATA ANALYSIS SOFTWARE AND SYSTEMS, 26., 2019. **Proceedings...** [S.l.], 2019. v. 521, p. 515. 3

GROPP, W.; LUSK, E. **User's guide for MPICH, a portable implementation of MPI**. [S.l.: s.n.], 1996. OSTI 378910. DOI 10.2172/378910. 7

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. 3

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**, v. 15, n. 7, p. 475–476, 2018. DOI 10.1038/s41592-018-0046-7. 118

GUELTON, S. Pythran: crossing the Python frontier. **Computing in Science & Engineering**, v. 20, n. 2, p. 83–89, 2018. DOI 10.1109/MCSE.2018.021651342. 126

GUELTON, S.; BRUNET, P.; AMINI, M.; MERLINI, A.; CORBILLON, X.; RAYNAUD, A. Pythran: enabling static optimization of scientific Python programs. **Computational Science & Discovery**, v. 8, n. 1, p. 014001, 2015. DOI 10.1088/1749-4680/8/1/014001. 126

HINSEN, K. The molecular modeling toolkit: a case study of a large scientific application in Python. In: INTERNATIONAL PYTHON CONFERENCE, 6., 1997. **Proceedings...** 1997. Available from: <`http://legacy.{P}ython.org/workshops/1997-10/proceedings/hinsen.html`>. Access in: Dec 2021. 1

HOLD-GEOFFROY, Y.; GAGNON, O.; PARIZEAU, M. Once you SCOOP, no need to fork. In: CONFERENCE ON EXTREME SCIENCE AND ENGINEERING DISCOVERY ENVIRONMENT, 2014. **Proceedings...** [S.l.], 2014. p. 1–8. DOI 10.1145/2616498.2616565. 125

HOLM, H. H.; BRODTKORB, A. R.; SÆTRA, M. L. GPU computing with Python: performance, energy efficiency and usability. **Computation**, v. 8, n. 1, p. 4, 2020. DOI 10.3390/computation8010004. 24, 122

IEEE Spectrum. **Interactive: top programming languages 2021**. 2021. Available from: <`http://spectrum.ieee.org/top-programming-languages/`>. Access in: Dec 2021. 1, 2

IMAMBI, S.; PRAKASH, K. B.; KANAGACHIDAMBARESAN, G. **Programming with TensorFlow**. [S.l.]: Springer, 2021. ISBN 978-3-030-57079-8. 121

INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS (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. 2

KETKAR, N. **Introduction to PyTorch**. [S.l.]: Apress, Berkeley, CA, 2017. DOI 10.1007/978-1-4842-2766-4_12. 116, 121

KETKAR, N.; MOOLAYIL, J. **Introduction to Pytorch**. [S.l.]: Springer, 2021. 3

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. 122

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**, v. 38, n. 3, p. 157–174, 2012. DOI 10.1016/j.parco.2011.09.001. 122, 128

KNUTH, D. **Literate programming**. [S.l.]: Cambridge University Press, 1992. (Center for the Study of Language and Information Publication Lecture Notes). ISBN 9780937073803. 117

KOLESNIKOV, I.; MENDES, C.; CARVALHO, R. de; ROSA, R. Optimized galaxy bayesian surface photometry on a many-core platform. In: SYMPOSIUM ON HIGH PERFORMANCE COMPUTING SYSTEMS, 21., 2020. **Proceedings...** Porto Alegre, RS, Brasil: SBC, 2020. p. 335–346. Available from: <https://sol.sbc.org.br/index.php/wscad/ARTICLE/view/14081>. 17

KRAMER, O. **Machine learning for evolution strategies**. [S.l.]: Springer, 2016. 5

LAM, S. K.; PITROU, A.; SEIBERT, S. Numba: a LLVM-based Python JIT compiler. In: WORKSHOP ON THE LLVM COMPILER INFRASTRUCTURE IN HPC, 2., 2015, Austin, Texas. **Proceedings...** New York, NY, USA: Association for Computing Machinery, 2015. (LLVM '15). ISBN 9781450340052. DOI 10.1145/2833157.2833162. 10, 11, 115, 116

LAM, S. K.; SEIBERT, S. How to accelerate an existing codebase with Numba. In: ANNUAL SCIENTIFIC COMPUTING WITH PYTHON CONFERENCE, 18., 2019. **Proceedings...** SciPy, 2019. Available from: <http://www.scipy2019.scipy.org/talks-posters/How-to-Accelerate-an-Existing-Codebase-with-Numba>. Access in: Dec 2021. 12

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.). **Proceedings of the third international conference on high performance scientific computing**. [S.l.]: Springer, 2008. p. 337?357. ISBN 978-3-540-79409-7. DOI 10.1007/978-3-540-79409-7_23. 27

LATTNER, C.; ADVE, V. LLVM: a compilation framework for lifelong program analysis & transformation. In: INTERNATIONAL SYMPOSIUM ON CODE

GENERATION AND OPTIMIZATION, 2004. **Proceedings...** [S.l.]: CGO, 2004. p. 75–86. DOI 10.1109/CGO.2004.1281665. 10, 11

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. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING WORKSHOPS, 44., 2015. **Proceedings...** [S.l.], 2015. p. 61–66. DOI 10.1109/icppw.2015.44. 13

LIMPRASERT, W. Parallel random forest with IPython cluster. In: INTERNATIONAL COMPUTER SCIENCE AND ENGINEERING CONFERENCE, 2015. **Proceedings...** [S.l.], 2015. p. 1–6. 3, 8

LUNACEK, M.; BRADEN, J.; HAUSER, T. The scaling of many-task computing approaches in Python on cluster supercomputers. In: INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, 2013. **Proceedings...** [S.l.]: IEEE, 2013. p. 1–8. DOI 110.1109/CLUSTER.2013.6702678. 1

MAROWKA, A. On parallel software engineering education using Python. **Education and Information Technologies**, v. 23, n. 1, p. 357–372, 2018. DOI 10.1007/s10639-017-9607-0. 3, 10

_____. Python accelerators for high-performance computing. **The Journal of Supercomputing**, v. 74, n. 4, p. 1449–1460, 2018. DOI 10.1007/s11227-017-2213-5. 126

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. 14

MCLEOD, C. A framework for distributed deep learning layer design in Python. **CoRR**, 2015. ArXiv: 1510.07303 [cs.LG]. 125

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. 4

_____. Comparison of high performance computing approaches in the Python environment for a five-point stencil test problem. In: BRAZILIAN E-SCIENCE WORKSHOP; CONGRESS OF THE BRAZILIAN COMPUTER SOCIETY, 41., 2021. **Proceedings...** [S.l.], 2021. p. 33–40. ISSN 2763-8774. DOI 10.5753/bresci.2021.15786. 4

MORITZ, P. et al. Ray: a distributed framework for emerging AI applications. In: SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, 13., 2018. **Proceedings...** 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. 124

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. 16

NASCIMENTO, F. J. B. D.; ARANTES FILHO, L. R.; 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**, v. 11, n. 2, p. 31–41, may 2019. DOI 10.5335/rbca.v11i2.9037. 2

NISHINO, R.; LOOMIS, S. H. C. CuPy: a NumPy-compatible library for NVIDIA GPU calculations. In: INTERNATIONAL CONFERENCE ON NEURAL INFORMATION PROCESSING SYSTEMS, 31., 2017. **Proceedings...** [S.l.]: NIPS, 2017. p. 151. 17

ODEN, L. Lessons learned from comparing C-CUDA and Python-Numba for GPU-computing. In: EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING, 28., 2020. **Proceedings...** [S.l.], 2020. p. 216–223. DOI 10.1109/PDP50117.2020.00041. 123

OLIPHANT, T. E. Python for scientific computing. **Computing in Science & Engineering**, v. 9, n. 3, p. 10–20, 2007. DOI 10.1109/MCSE.2007.58. 127

OSMIALOWSKI, P. How the Flang frontend works: introduction to the interior of the open-source Fortran frontend for LLVM. In: WORKSHOP ON THE LLVM COMPILER INFRASTRUCTURE IN HPC, 4., 2017. **Proceedings...** [S.l.], 2017. p. 1–14. DOI 10.1145/3148173.3148183. 123

PALACH, J. **Parallel programming with Python**. [S.l.]: Packt Publishing Ltd, 2014. 116, 123

PÉREZ, F.; GRANGER, B. E. IPython: a system for interactive scientific computing. **Computing in Science & Engineering**, v. 9, n. 3, p. 21–29, 2007. 1

PERKEL, J. M. Why Jupyter is data scientists' computational Notebook of choice. **Nature**, v. 563, n. 7732, p. 145–147, 2018. DOI 10.1038/d41586-018-07196-1. 117

PETERSON, P. F2PY: a tool for connecting Fortran and Python programs. **International Journal of Computational Science and Engineering**, v. 4, n. 4, p. 296–305, 2009. DOI 10.1504/IJCSE.2009.029165. 3, 13, 116

RESCH, W. Python in HPC: profiling, optimizing, and executing Python in high performance computing. **National Institutes of Health (NIH) High Performance Computing (HPC) Group. U.S. Department of Health & Human Services**, 2020. Slides and Handouts from Previous HPC Classes. Classes, Seminars & Walk-In Consults. Available from: <https://hpc.nih.gov/>. Access in: Dec 2021. 21

ROCKLIN, M. Dask: parallel computation with blocked algorithms and task scheduling. In: PYTHON IN SCIENCE CONFERENCE, 14., 2015. **Proceedings...** [S.l.], 2015. v. 126, p. 126–132. DOI 10.25080/Majora-7b98e3ed-013. 116, 123

_____. **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. 123

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. PMID 10660911. Available from: <http://pubmed.ncbi.nlm.nih.gov/10660911/>. Access in: Dec 2021. 113, 114

SCHULZ, R. **3D FFT with 2D decomposition**. Knoxville: The University of Tennessee, 2008. CS Project report. Center for molecular biophysics. 34

SCULLIN, W.; BELHORN, M.; THOMAS, R. Python for high performance computing. In: EXASCALE COMPUTING PROJECT, 2., 2018. **Proceedings...** 2018. William Scullin, Assistant Computational Scientist, ALCF, ANL. Matt Belhorn, OLCF, ORNL. Rollin Thomas, Big Data Architect, NERSC, LBNL. Available from: <https://www.ecpannualmeeting.com/>. Access in: Dec 2021. 21

SEBESTA, R. W.; MUKHERJEE, S.; BHATTACHARJEE, A. K. **Concepts of programming languages**. [S.l.]: Boston: Pearson, 2016. ISBN 9780133943023. 113

SEHRISH, S.; KOWALKOWSKI, J.; PATERNO, M.; GREEN, C. Python and HPC for high energy physics data analyses. In: WORKSHOP ON PYTHON FOR HIGH-PERFORMANCE AND SCIENTIFIC COMPUTING, 7., 2017.

Proceedings... New York, NY, USA: Association for Computing Machinery, 2017. (PyHPC'17). ISBN 9781450351249. DOI 10.1145/3149869.3149877. 1, 23

SINGH, N.; BROWNE, L.-M.; BUTLER, R. Parallel astronomical data processing with Python: recipes for multicore machines. **Astronomy and Computing**, v. 2, p. 1–10, 2013. DOI 10.1016/j.ascom.2013.04.002. 126

SINGH, S. Declarative data-parallel programming with the accelerator system. In: SIGPLAN WORKSHOP ON DECLARATIVE ASPECTS OF MULTICORE PROGRAMMING, 5., 2010. **Proceedings...** [S.l.], 2010. p. 1–2. DOI 10.1145/1708046.1708048. 113

SOUZA, C. R. d.; PANETTA, J.; STEPHANY, S. Analysis of communication performance using MPI 3.0 shared memory functionality (in Portuguese). **Revista Cereus**, v. 10, n. 2, p. 193–210, 2018. ISSN 2175-7275. DOI 10.18605/2175-7275/cereus.v10n2p193-210. 27

SOUZA, J.; MENDES, C.; SANTOS, R. Performance optimization of the brazil solar radiation model (in Portuguese). In: ERAD; REGIONAL SCHOOL OF HPC, 9., 2018, São Paulo. **Proceedings...** [S.l.]: Brazilian Computer Society (SBC), 2018. DOI 10.5753/eradsp.2018.13601. 2

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. 7

VAN ROSSUM, G. Python programming language. In: USENIX ANNUAL TECHNICAL CONFERENCE, 2007. **Proceedings of the 2007 Annual Technical Conference**. 2007. v. 41, p. 36. Available from: <http://www.usenix.org/conference/2007-usenix-annual-technical-conference/presentation/{P}ython-programming-language>. Access in: Dec 2021. 127

VASILEV, V.; CANAL, P.; NAUMANN, A.; RUSSO, P. Cling – the new interactive interpreter for ROOT 6. **Journal of Physics: Conference Series**, v. 396, n. 5, 2012. DOI 10.1088/1742-6596/396/5/052071. 114

VINCENT, P.; WITHERDEN, F.; VERMEIRE, B.; PARK, J. S.; IYER, A. Towards green aviation with Python at petascale. In: INTERNATIONAL

CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2016. **Proceedings...** 2016. p. 1–11. ISBN 978-1-4673-8815-3. ISSN 2167-4337. DOI 10.1109/SC.2016.1. Available from: <`https://ieeexplore.ieee.org/document/7876999`>. Access in: Dec 2021. 22

VIRTANEN, P. et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. **Nature Methods**, v. 17, n. 3, p. 261–272, feb. 2020. DOI 10.1038/s41592-019-0686-2. 1, 6

WAGNER, M.; LLORT, G.; MERCADAL, E.; GIMÉNEZ, J.; LABARTA, J. Performance analysis of parallel Python applications. **Procedia Computer Science**, v. 108, p. 2171–2179, 2017. 23

WALT, S. V. D.; COLBERT, S. C.; VAROQUAUX, G. The NumPy array: a structure for efficient numerical computation. **Computing in Science & Engineering**, v. 13, n. 2, p. 22–30, 2011. DOI 10.1109/MCSE.2011.37. 7, 116

WITHERDEN, F.; FARRINGTON, A.; VINCENT, P. PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. **Computer Physics Communications**, v. 185, n. 11, p. 3028–3040, 2014. ISSN 0010-4655. DOI 10.1016/j.cpc.2014.07.011. Available from: <`https://www.sciencedirect.com/science/ARTICLE/pii/S0010465514002549`>. 22

YEAGER, L.; BERNAUER, J.; GRAY, A.; HOUSTON, M. Digits: the deep learning GPU training system. In: ICML AUTOML WORKSHOP, 2015. **Proceedings...** 2015. Available from: <`http://indico.ijclab.in2p3.fr/event/2914/contributions/6476/subcontributions/170`>. Access in: Dec 2021. 122

ZIOGAS, A. N.; SCHNEIDER, T.; BEN-NUN, T.; CALOTOIU, A.; De Matteis, T.; LICHT, J. de F.; LAVARINI, L.; HOEFLER, T. Productivity, portability, performance: data-centric Python. In: INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2021. **Proceedings...** [S.l.], 2021. p. 1–13. 23

APPENDIX A - PUBLISHED ARTICLES

Two 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 test:

- 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.

  Abstract: Several of the most important high-performance computing approaches available in the Python programming environment of the LNCC Santos Dumont supercomputer, are compared using a specific test problem. Python includes specific libraries, implementations, development tools, documentation, optimization and parallelization resources. It provides a straightforward way to program using a high level of abstraction, but the parallelization features for exploring multiple cores, processors, or accelerators such as GPUs, are diverse and may not be easily chosen by the user. Serial and parallel implementations of a test problem in Fortran 90 are taken as benchmarks to compare performance. This work is a primer for the use of HPC resources in Python.

- 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.

  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, doc-

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

ORCID

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

APPENDIX B - PYTHON ENVIRONMENT 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).

B.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 to write code following a functional declarative programming paradigm when in combination with the user 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).

B.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. Thus, using Python with libraries that allow JIT compilation combines the advantages of both languages, namely the readable syntax of a modern interpreted language with the better performance of a compiled language. An example of a JIT interpreter for Python is Pypy (BIHAM; SEBERRY, 2006).

B.3 Python intermediate representation

During the executing of 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 form. After the 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).

B.4 Python decorators

Python treats classes and functions as objects, allowing 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 Numba JIT compiler, allow the definition of functions that must be compiled only at runtime. 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).

B.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). Python libraries are not necessarily written using the Python language, and in fact the interfacing and extension capability is used to create libraries from various sources, and within the library there can be optimized native code that is executed directly by the processor without being interpreted, making the library as fast as a compiled language. Consequently, one of the ways to get performance in Python is to use optimized libraries, especially in computationally intensive parts.

There are also compilers, such as Cython, that can create standard Python libraries by compiling the source code into C code, which in turn is built by a C compiler to generate optimized native code, and the end result is a library (BEHNEL et al., 2010). Another example is the Numba compiler and library, which allows to compile just-in-time (during runtime) a subset of the Python language and the NumPy library (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).

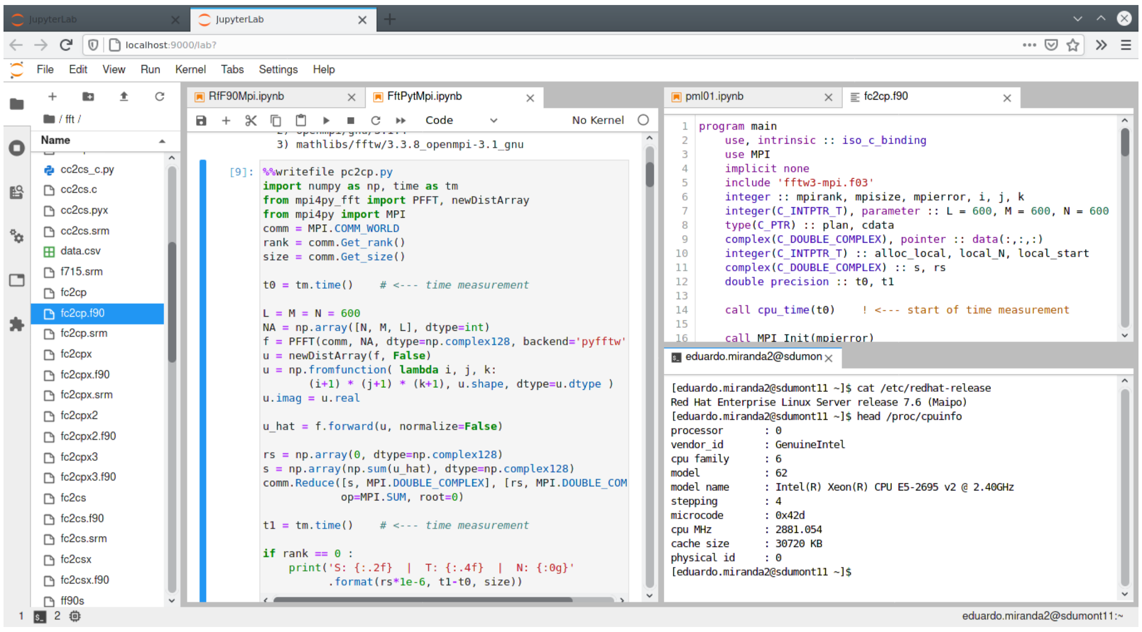
Throughout this text, for the sake of simplicity, each core of a multi-core processor will be denoted as CPU, and each general-purpose graphics accelerator card (GPGPU - General Purpose Graphics Processing Unit) will be denoted as GPU. In addition, High Performance Computing, which obviously includes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

B.6 JupyterLab (and Jupyter Notebook)

JupyterLab, which has been used in all implementations in this work, is a user interface that is an evolution, is compatible, and offers all the familiar building blocks of classic Jupyter Notebook. JupyterLab is an interactive development environment that has the flexibility to allow combination of executable code snippets to solve a problem, with explanatory text, and calculation results including graphics, in addition to having an interactive authoring application running in the web browser. It offers some features of literate programming, which is a programming paradigm introduced by Donald Knuth, in which code accompanies an explanation of its logic in a natural language interspersed with snippets of source code that can be compiled and/or executed (KNUTH, 1992). The approach is used in scientific computing and data science for reproducible research and open access purposes to facilitate understanding. 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 client-server architecture with a communication protocol that allows running servers on remote machines (a cluster, or a supercomputer), and interactively developing a prototype on a local laptop, while executing the code transparently on remote machine (Figure B.1).

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 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 or environments, for example, it is possible to develop code in F90, C, R, Bash shell, and others. It is also possible to access the operating system shell and perform most of the tasks

Figure B.1 - JupyterLab interface.



```
  3) mathlibs/fftw/3.3.8_openmpi-3.1_gnu

[9]: %%writefile pc2cp.py
import numpy as np, time as tm
from mpi4py_fft import PFFT, newDistArray
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

t0 = tm.time()    # <--- time measurement

L = M = N = 600
NA = np.array([N, M, L], dtype=int)
f = PFFT(comm, NA, dtype=np.complex128, backend='pyfftw'
u = newDistArray(f, False)
u = np.fromfunction( lambda i, j, k:
       (i+1) * (j+1) * (k+1), u.shape, dtype=u.dtype )
u.imag = u.real

u_hat = f.forward(u, normalize=False)

rs = np.array(0, dtype=np.complex128)
s = np.array(np.sum(u_hat), dtype=np.complex128)
comm.Reduce([s, MPI.DOUBLE_COMPLEX], [rs, MPI.DOUBLE_COM
            op=MPI.SUM, root=0)

t1 = tm.time()    # <--- time measurement

if rank == 0 :
    print('S: {:.2f}  |  T: {:.4f}  |  N: {:0g}'
          .format(rs*1e-6, t1-t0, size))
```

```
program main
    use, intrinsic :: iso_c_binding
    use MPI
    implicit none
    include 'fftw3-mpi.f03'
    integer :: mpirank, mpisize, mpierror, i, j, k
    integer(C_INTPTR_T), parameter :: L = 600, M = 600, N = 600
    type(C_PTR) :: plan, cdata
    complex(C_DOUBLE_COMPLEX), pointer :: data(:,:,:)
    integer(C_INTPTR_T) :: alloc_local, local_N, local_start
    complex(C_DOUBLE_COMPLEX) :: s, rs
    double precision :: t0, t1

    call cpu_time(t0)    ! <--- start of time measurement

    call MPI_Init(mpierror)
```

```
[eduardo.miranda2@sdumont11 ~]$ cat /etc/redhat-release
Red Hat Enterprise Linux Server release 7.6 (Maipo)
[eduardo.miranda2@sdumont11 ~]$ head /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 62
model name      : Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
stepping        : 4
microcode       : 0x42d
cpu MHz         : 2881.054
cache size      : 30720 KB
physical id     : 0
[eduardo.miranda2@sdumont11 ~]$
```

Source: Author's production.

and features from the command line. It also creates or edits a standalone document on the local machine containing a Notebook, which later can also be viewed in an appropriate document reader, or the document can be exported to other formats such as pdf, html, and LaTeX. This document can be shared and also used in another Notebook session. It is possible to use Notebook in order to store the entire history with the step by step of what was done during development, including the outputs and results of the remote machine, interspersed with executable code snippets, and with the description of what was done, in separate cells. This Notebook may be reproducible, cells may be re-executed, and the same results may be obtained by mixing documentation with executable code, as long as it is on the same machine or in the same configuration, or else the Notebook may be viewed as just a regular document, and comments may be be added as they are machine independent. 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 to install, run, and update Python packages and resolve their dependencies, including installing packages with the same versions used during code development (BISONG, 2019).

B.7 Conda package manager and ecosystem

Conda (GRÜNING et al., 2018) is a free, open source, multi-featured Python package and environment manager that is cross-platform and can work with languages other than Python. As an example, it is used by the Python Anaconda distribution. Environment management is one of the main aspects of Conda, allowing different versions of packages to be used, or even completely separate package installations. It is possible to work with different Python environments, which can be easily created, saved, loaded and switched. Other features are conflict resolution management, package dependencies, and package breaking. In this work, several features of Conda were used, such as nested activation to allow two environments to be used at the same time. Conda also has features to list current packages and install on another machine to allow reproducibility. It is also possible to create stacked environments, to allow, for example, adding packages to an existing environment without modifying it, and this feature can be useful, for example, in SDumont if the existing Python distribution being used does not have the JupyterLab package, or another package. A new stacked environment is created, in the user area, to add the missing packages, without modifying the environment that already exists. Conda also has a large online package repository.

APPENDIX C - OTHER PYTHON HPC APPROACHES

This appendix briefly describes other approaches that were not directly used in this work, but are somehow related, or were used as a reference. Perhaps the most employed of these approaches is PyTorch. Most of these solutions are actively being developed and offer some form of high-performance or parallel processing, multi-processing, or interfacing capabilities with C or F90. Furthermore, it does not include cloud computing, grid computing and related projects such as distributed file systems.

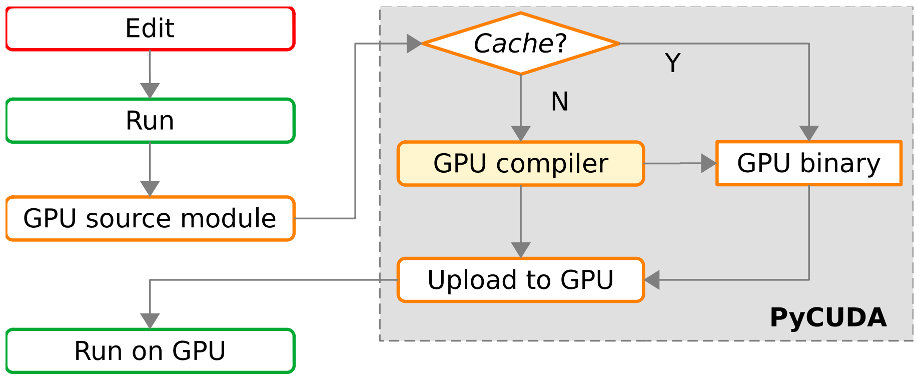
C.1 PyTorch

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; IMAMBI et al., 2021). 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., 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).

C.2 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

Figure C.1 - PyCuda JIT interpretation and compilation diagram.



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

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). Figure C.1 shows the JIT PyCuda interpretation and compilation diagram, which includes an interpreted phase for tests and execution on the CPU, and a compilation phase for the GPU that uses a temporary storage of binary codes, to avoid repeated compilations at each run.

C.3 Other Nvidia GPU supports for Python

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).

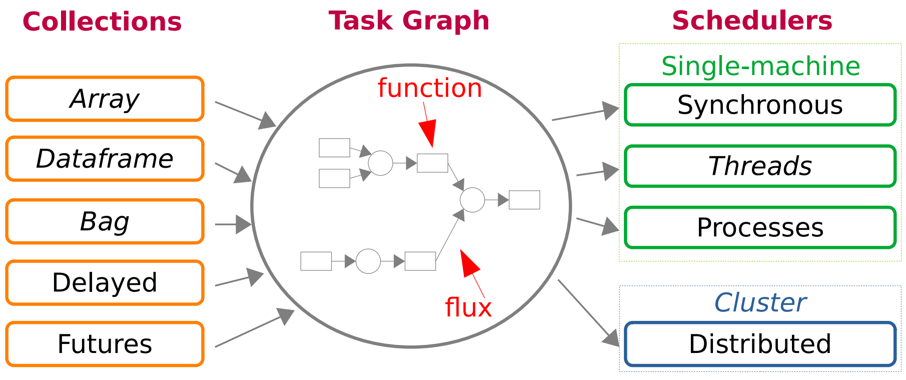
C.4 Dask for Python

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 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, thus allowing efficient parallelization to be used. The Anaconda distribution is the most popular of the Python versions for Data Science. 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 C.2. This allows to identify independent tasks, which correspond to the functions of each node, and which allow 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).

C.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).

Figure C.2 - Diagram of the task graph and Dask schedulers.



Source: Adapted from dask.org (2022).

C.6 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).

C.7 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).

C.8 Ray

System for building and running distributed applications, supporting GPU, multi-processing, 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).

C.9 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).

C.10 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).

C.11 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).

C.12 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).

C.13 Pyro

Library that allows 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).

C.14 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 to work with systems like Hadoop (distributed computing) and cloud computing, among others (DRABAS; LEE, 2017).

C.15 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).

C.16 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 to easily use the data parallelism paradigm (SINGH et al., 2013).

C.17 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).

C.18 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. It is also possible to generate calls to OpenMP (library for shared memory multi-process programming), and use SIMD instructions (GUELTON et al., 2015; GUELTON, 2018).

C.19 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, 2007).

C.20 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).

C.21 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 such as the Python fast development environment, and also writing a simpler code, since 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).

C.22 PyOpenCL

PyOpenCL works as a wrapper for OpenCL which is a framework for writing code for heterogeneous platforms consisting of CPUs, GPUs, DSPs, FPGAs and other processors or hardware accelerators. It is a library API to allow Python to access it. OpenCL was developed in C++ and allows 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).

APPENDIX D - CODES IMPLEMENTED IN THIS WORK

This appendix shows the main codes of the three test cases shown in this work. The full set of codes is available at the repository `http://github.com/efurlanm/msc22`.

D.1 Implementations of the stencil test case

D.1.1 Serial F90

Listing D.1 - Serial F90 implementation of the stencil test case.

```fortran
program stencil
    implicit none
    integer, parameter  :: nsources=3
    integer             :: n=4800     ! nxn grid (4800)
    integer             :: energy=1   ! energy to be injected per iteration (1)
    integer             :: niters=500 ! number of iterations (500)
    integer             :: iters, i, j, size, sizeStart, sizeEnd
    integer, dimension(3, 2)        :: sources
    double precision, allocatable   :: aold(:,:), anew(:,:)
    double precision    :: t=0.0, t1=0.0, heat=0.0

    call cpu_time(t1)
    t = -t1

    size = n + 2
    sizeStart = 2
    sizeEnd = n + 1

    allocate(aold(size, size))
    allocate(anew(size, size))
    aold = 0.0
    anew = 0.0

    sources(1,:) = (/ n/2,   n/2   /)
    sources(2,:) = (/ n/3,   n/3   /)
    sources(3,:) = (/ n*4/5, n*8/9 /)

    do iters = 1, niters, 2

        do j = sizeStart, sizeEnd
            do i = sizeStart, sizeEnd
                anew(i,j) = aold(i,j)/2.0 + (aold(i-1,j) + aold(i+1,j) +  &
                            aold(i,j-1) + aold(i,j+1))/4.0/2.0
            enddo
        enddo

        do i = 1, nsources
            anew(sources(i,1)+1, sources(i,2)+1) =  &
                anew(sources(i,1)+1, sources(i,2)+1) + energy
        enddo

        do j = sizeStart, sizeEnd
            do i = sizeStart, sizeEnd
                aold(i,j) = anew(i,j)/2.0 + (anew(i-1,j) + anew(i+1,j) +  &
                            anew(i,j-1) + anew(i,j+1))/4.0/2.0
            enddo
        enddo

        do i = 1, nsources
            aold(sources(i,1)+1, sources(i,2)+1) =  &
                aold(sources(i,1)+1, sources(i,2)+1) + energy
        enddo

    enddo

    heat = 0.0
    do j = sizeStart, sizeEnd
        do i = sizeStart, sizeEnd
            heat = heat + aold(i,j)
        end do
    end do

    deallocate(aold)
    deallocate(anew)

    call cpu_time(t1)
    t = t + t1

    write(*, "('Heat = ' f0.4' | ')", advance="no") heat
    write(*, "('Time = 'f0.4)") t

end
```

D.1.2 Parallel F90

Listing D.2 - Parallel F90 implementation of the stencil test case.

```fortran
program stencil
    use MPI
    implicit none
    integer :: n=0          ! nxn grid
    integer :: energy=0     ! energy to be injected per iteration
    integer :: niters=0     ! number of iterations
    integer :: iters, i, j, px, py, rx, ry
    integer :: north, south, west, east, bx, by, offx, offy
    integer :: nargs=0, iargc
    integer :: mpirank, mpisize, mpitag=1, mpierror
    integer, dimension(3) :: args
    integer, dimension(2) :: pdims=0
    integer, dimension(4) :: sendrequest, recvrequest
    double precision :: mpiwtime=0.0, heat=0.0, rheat=0.0
    double precision, dimension(:), allocatable   :: sendnorthgz, sendsouthgz
    double precision, dimension(:), allocatable   :: recvnorthgz, recvsouthgz
    double precision, dimension(:,:), allocatable :: aold, anew
    character(len=50)                             :: argv

    integer, parameter  :: nsources=3        ! three heat sources
    ! locnsources = number of sources in my area
    integer             :: locnsources=0, locx, locy
    ! locsources = sources local to my rank
    integer, dimension(nsources, 2) :: locsources=0, sources

    call MPI_Init(mpierror)
    call MPI_Comm_rank(MPI_COMM_WORLD, mpirank, mpierror)
    call MPI_Comm_size(MPI_COMM_WORLD, mpisize, mpierror)

    if (mpirank == 0) then                      ! rank 0 argument checking
        mpiwtime = -MPI_Wtime()
        nargs = iargc()
        call getarg(1, argv); read(argv, *) n       ! nxn grid
        call getarg(2, argv); read(argv, *) energy  ! energy to be injected
        call getarg(3, argv); read(argv, *) niters  ! number of iterations
        args = [ n, energy, niters ]                ! distribute arguments
        call MPI_Bcast(args, 3, MPI_INTEGER, 0, MPI_COMM_WORLD, mpierror)
    else
        call MPI_Bcast(args, 3, MPI_INTEGER, 0, MPI_COMM_WORLD, mpierror)
        n = args(1); energy = args(2); niters  = args(3)
    endif

    ! Creates a division of processors in a Cartesian grid
    ! MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
    !   NNODES - number of nodes in a grid
    !   NDIMS - number of Cartesian dimensions
    !   DIMS - array specifying the number of nodes in each dimension
    ! Examples:
```

```
    !   MPI_Dims_create(6, 2, dims)  ->  (3,2)
    !   MPI_Dims_create(7, 2, dims)  ->  (7,1)
    call MPI_Dims_create(mpisize, 2, pdims, mpierror)

    ! determine my coordinates (x,y)
    px = pdims(1)
    py = pdims(2)
    rx = mod(mpirank, px)
    ry = mpirank / px

    ! determine my four neighbors
    north = (ry - 1) * px + rx; if( (ry - 1) < 0  ) north = MPI_PROC_NULL
    south = (ry + 1) * px + rx; if( (ry + 1) >= py) south = MPI_PROC_NULL
    west = ry * px + rx - 1;    if( (rx - 1) < 0  ) west  = MPI_PROC_NULL
    east = ry * px + rx + 1;    if( (rx + 1) >= px) east  = MPI_PROC_NULL

    ! decompose the domain
    bx = n / px             ! block size in x
    by = n / py             ! block size in y
    offx = (rx * bx) + 1    ! offset in x
    offy = (ry * by) + 1    ! offset in y

    ! initialize heat sources
    sources = reshape( [ n/2,   n/2,        &
                         n/3,   n/3,        &
                         n*4/5, n*8/9 ],    &
              shape(sources), order=[2, 1])

    do i = 1, nsources      ! determine which sources are in my patch
        locx = sources(i, 1) - offx
        locy = sources(i, 2) - offy
        if(locx >= 0 .and. locx <= bx .and. locy >= 0 .and. locy <= by) then
            locnsources = locnsources + 1
            locsources(locnsources, 1) = locx + 2
            locsources(locnsources, 2) = locy + 2
        endif
    enddo

    ! allocate communication buffers
    allocate(sendnorthgz(bx))   ! send buffers
    allocate(sendsouthgz(bx))
    allocate(recvnorthgz(bx))   ! receive buffers
    allocate(recvsouthgz(bx))
    ! allocate two work arrays
    allocate(aold(bx+2, by+2)); aold = 0.0   ! 1-wide halo zones!
```

```fortran
    allocate(anew(bx+2, by+2)); anew = 0.0   ! 1-wide halo zones!

    ! laco principal das iteracoes
    do iters = 1, niters, 2

        ! --- anew <- stencil(aold) ---
        if(north /= MPI_PROC_NULL) then
            sendnorthgz = aold(2, 2:bx+1)
            recvnorthgz = 0.0
            call MPI_IRecv(recvnorthgz, bx, MPI_DOUBLE_PRECISION, north,  &
                           mpitag, MPI_COMM_WORLD, recvrequest(1), mpierror)
            call MPI_ISend(sendnorthgz, bx, MPI_DOUBLE_PRECISION, north,  &
                           mpitag, MPI_COMM_WORLD, sendrequest(1), mpierror)
        endif
        if(south /= MPI_PROC_NULL) then
            sendsouthgz = aold(bx+1, 2:bx+1)
            recvsouthgz(:) = 0.0
            call MPI_IRecv(recvsouthgz, bx, MPI_DOUBLE_PRECISION, south,  &
                           mpitag, MPI_COMM_WORLD, recvrequest(2), mpierror)
            call MPI_ISend(sendsouthgz, bx, MPI_DOUBLE_PRECISION, south,  &
                           mpitag, MPI_COMM_WORLD, sendrequest(2), mpierror)
        endif
        if(east /= MPI_PROC_NULL) then
            call MPI_IRecv(aold(2:bx+1, bx+2), bx, MPI_DOUBLE_PRECISION, east, &
                           mpitag, MPI_COMM_WORLD, recvrequest(3), mpierror)
            call MPI_ISend(aold(2:bx+1, bx+1), bx, MPI_DOUBLE_PRECISION, east, &
                           mpitag, MPI_COMM_WORLD, sendrequest(3), mpierror)
        endif
        if(west /= MPI_PROC_NULL) then
            call MPI_IRecv(aold(2:bx+1, 1), bx, MPI_DOUBLE_PRECISION, west, &
                          mpitag, MPI_COMM_WORLD, recvrequest(4), mpierror)
            call MPI_ISend(aold(2:bx+1, 2), bx, MPI_DOUBLE_PRECISION, west, &
                          mpitag, MPI_COMM_WORLD, sendrequest(4), mpierror)
            endif
        if(north /= MPI_PROC_NULL) then
            call MPI_Wait(recvrequest(1), MPI_STATUS_IGNORE, mpierror)
            call MPI_Wait(sendrequest(1), MPI_STATUS_IGNORE, mpierror)
            aold(1, 2:bx+1)=recvnorthgz
        endif
        if(south /= MPI_PROC_NULL) then
            call MPI_Wait(recvrequest(2), MPI_STATUS_IGNORE, mpierror)
            call MPI_Wait(sendrequest(2), MPI_STATUS_IGNORE, mpierror)
            aold(bx+2, 2:bx+1)=recvsouthgz
        endif
        if(east /= MPI_PROC_NULL) then
```

```fortran
            call MPI_Wait(recvrequest(3), MPI_STATUS_IGNORE, mpierror)
            call MPI_Wait(sendrequest(3), MPI_STATUS_IGNORE, mpierror)
        endif
        if(west /= MPI_PROC_NULL) then
            call MPI_Wait(recvrequest(4), MPI_STATUS_IGNORE, mpierror)
            call MPI_Wait(sendrequest(4), MPI_STATUS_IGNORE, mpierror)
        endif

        ! update grid points
        do j = 2, by+1
            do i = 2, bx+1
                anew(i, j) = aold(i, j)/2.0 + (aold(i-1, j) + aold(i+1, j) +  &
                             aold(i, j-1) + aold(i, j+1)) / 4.0 / 2.0
            enddo
        enddo

        do i = 1, locnsources
            anew(locsources(i, 1), locsources(i, 2)) =   &
                anew(locsources(i, 1), locsources(i, 2)) + energy
        enddo

        ! --- aold <- stencil(anew) ---
        if(north /= MPI_PROC_NULL) then
            sendnorthgz=anew(2, 2:bx+1)
            call MPI_IRecv(recvnorthgz, bx, MPI_DOUBLE_PRECISION, north, mpitag,
     &
                            MPI_COMM_WORLD, recvrequest(1), mpierror)
            call MPI_ISend(sendnorthgz, bx, MPI_DOUBLE_PRECISION, north, mpitag,
     &
                            MPI_COMM_WORLD, sendrequest(1), mpierror)
        endif
        if(south /= MPI_PROC_NULL) then
            sendsouthgz=anew(bx+1, 2:bx+1)
            call MPI_IRecv(recvsouthgz, bx, MPI_DOUBLE_PRECISION, south, mpitag,
     &
                            MPI_COMM_WORLD, recvrequest(2), mpierror)
            call MPI_ISend(sendsouthgz, bx, MPI_DOUBLE_PRECISION, south, mpitag,
     &
                            MPI_COMM_WORLD, sendrequest(2), mpierror)
        endif
        if(east /= MPI_PROC_NULL) then
            call MPI_IRecv(anew(2:bx+1, bx+2), bx, MPI_DOUBLE_PRECISION, east,  &
                            mpitag, MPI_COMM_WORLD, recvrequest(3), mpierror)
            call MPI_ISend(anew(2:bx+1, bx+1), bx, MPI_DOUBLE_PRECISION, east,  &
                            mpitag, MPI_COMM_WORLD, sendrequest(3), mpierror)
```

```fortran
    endif
    if(west /= MPI_PROC_NULL) then
        call MPI_IRecv(anew(2:bx+1, 1), bx, MPI_DOUBLE_PRECISION, west,
mpitag,  &
                        MPI_COMM_WORLD, recvrequest(4), mpierror)
        call MPI_ISend(anew(2:bx+1, 2), bx, MPI_DOUBLE_PRECISION, west,
mpitag,  &
                        MPI_COMM_WORLD, sendrequest(4), mpierror)
    endif
    if(north /= MPI_PROC_NULL) then
        call MPI_Wait(recvrequest(1), MPI_STATUS_IGNORE, mpierror)
        call MPI_Wait(sendrequest(1), MPI_STATUS_IGNORE, mpierror)
        anew(1, 2:bx+1)=recvnorthgz
    endif
    if(south /= MPI_PROC_NULL) then
        call MPI_Wait(recvrequest(2), MPI_STATUS_IGNORE, mpierror)
        call MPI_Wait(sendrequest(2), MPI_STATUS_IGNORE, mpierror)
        anew(bx+2, 2:bx+1)=recvsouthgz
    endif
    if(east /= MPI_PROC_NULL) then
        call MPI_Wait(recvrequest(3), MPI_STATUS_IGNORE, mpierror)
        call MPI_Wait(sendrequest(3), MPI_STATUS_IGNORE, mpierror)
    endif
    if(west /= MPI_PROC_NULL) then
        call MPI_Wait(recvrequest(4), MPI_STATUS_IGNORE, mpierror)
        call MPI_Wait(sendrequest(4), MPI_STATUS_IGNORE, mpierror)
    endif

    ! update grid points
    do j = 2, by+1
        do i = 2, bx+1
            aold(i, j) = anew(i, j)/2.0 + (anew(i-1, j) + anew(i+1, j) +  &
                         anew(i, j-1) + anew(i, j+1)) / 4.0 / 2.0
        enddo
    enddo

    do i = 1, locnsources
        aold(locsources(i, 1), locsources(i, 2)) =  &
            aold(locsources(i, 1), locsources(i, 2)) + energy
    enddo

enddo

! ALL REDUCE:
heat = 0.0
```

```fortran
    do j = 2, by+1
        do i = 2, bx+1
            heat = heat + aold(i, j)
        enddo
    enddo
    call MPI_Allreduce(heat, rheat, 1, MPI_DOUBLE_PRECISION, MPI_SUM,  &
                       MPI_COMM_WORLD, mpierror)

    if(mpirank == 0) then
        mpiwtime = mpiwtime + MPI_Wtime()
        write(*, "('Heat='     f0.2' | ')", advance="no") rheat
        write(*, "('Time='    f0.4' | ')", advance="no") mpiwtime
        write(*, "('MPI_Size=' i0  ' | ')", advance="no") mpisize
        write(*, "('MPI_Dims=('i0','i0') | ')", advance="no") pdims
        write(*, "('bx,by=('i0','i0')')") bx,by
    endif

    call MPI_Finalize(mpierror)
end
```

D.1.3 Serial F2PY

Listing D.3 - Serial F2PY implementation of the stencil test case - F90 code.

```fortran
subroutine st(n, energy, niters, heat, t)
    integer, intent(in) :: n, energy, niters
    double precision, intent(out) :: heat, t
    integer, parameter :: nsources=3
    integer :: iters, i, j, x, y, size, sizeStart, sizeEnd
    integer, dimension(3, 2) :: sources
    double precision, allocatable :: aold(:,:), anew(:,:)
    double precision :: t1=0.0, t2=0.0

    call cpu_time(t1)

    size = n + 2
    sizeStart = 2
    sizeEnd = n + 1

    allocate(aold(size, size))
    allocate(anew(size, size))
    aold = 0.0
    anew = 0.0

    sources(1,:) = (/ n/2,   n/2   /)
    sources(2,:) = (/ n/3,   n/3   /)
    sources(3,:) = (/ n*4/5, n*8/9 /)

    do iters = 1, niters, 2
        do j = sizeStart, sizeEnd
            do i = sizeStart, sizeEnd
                anew(i,j) = aold(i,j)/2.0 + (aold(i-1,j) + aold(i+1,j) +  &
                            aold(i,j-1) + aold(i,j+1)) / 4.0 / 2.0
            enddo
        enddo
        do i = 1, nsources
            x = sources(i,1) + 1
            y = sources(i,2) + 1
            anew(x,y) =  anew(x,y) + energy
        enddo
        do j = sizeStart, sizeEnd
            do i = sizeStart, sizeEnd
                aold(i,j) = anew(i,j)/2.0 + (anew(i-1,j) + anew(i+1,j) +  &
                            anew(i,j-1) + anew(i,j+1)) / 4.0 / 2.0
            enddo
        enddo
        do i = 1, nsources
            x = sources(i,1) + 1
            y = sources(i,2) + 1
            aold(x,y) = aold(x,y) + energy
        enddo
    enddo
    heat = 0.0
    do j = sizeStart, sizeEnd
        do i = sizeStart, sizeEnd
            heat = heat + aold(i,j)
        end do
    end do
    deallocate(aold)
    deallocate(anew)
    call cpu_time(t2)
    t = t2 - t1
end subroutine
```

Listing D.4 - Serial F2PY implementation of the stencil test case - Python code.

```python
from time import time
tp = time()
import stencil_f2py_seq
import numpy as np
```

```python

n             = 4800    # nxn grid; 4800,1,500->1500; 100,1,10->30; [4800]
energy        = 1       # energy to be injected per iteration; [1]
niters        = 500     # number of iterations; [500]

heat, t = stencil_f2py_seq.st(n, energy, niters)
tp = time() - tp

print("Heat = %0.4f | Time = %0.4f | TimePyt = %0.4f" %(heat, t, tp))
```

D.1.4 Parallel F2PY

Listing D.5 - Parallel F2PY implementation of the stencil test case - F90 module code.

```fortran
subroutine stm(n, energy, niters, oheat, otime, orank)
    use MPI
    implicit none
    integer, intent(in) :: n, energy, niters
    double precision, intent(out) :: oheat, otime
    integer, intent(out) :: orank

    integer :: iters, i, j, px, py, rx, ry
    integer :: north, south, west, east, bx, by, offx, offy
    integer :: mpirank, mpisize, mpitag=1, mpierror
    integer, dimension(2) :: pdims=0
    integer, dimension(4) :: sendrequest, recvrequest
    double precision :: mpiwtime=0.0, heat=0.0, rheat=0.0
    double precision, dimension(:), allocatable   :: sendnorthgz, sendsouthgz
    double precision, dimension(:), allocatable   :: recvnorthgz, recvsouthgz
    double precision, dimension(:,:), allocatable :: aold, anew

    integer, parameter  :: nsources=3        ! three heat sources
    ! locnsources = number of sources in my area
    integer             :: locnsources=0, locx, locy
    ! locsources = sources local to my rank
    integer, dimension(nsources, 2) :: locsources=0, sources

    call MPI_Init(mpierror)
    call MPI_Comm_rank(MPI_COMM_WORLD, mpirank, mpierror)
    call MPI_Comm_size(MPI_COMM_WORLD, mpisize, mpierror)

    if (mpirank == 0) then
        mpiwtime = -MPI_Wtime()     ! inicializa contador de Time
    endif

    ! Creates a division of processors in a Cartesian grid
    ! MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
    !   NNODES - number of nodes in a grid
    !   NDIMS - number of Cartesian dimensions
    !   DIMS - array specifying the number of nodes in each dimension
    ! Examples:
    !   MPI_Dims_create(6, 2, dims)  ->  (3,2)
    !   MPI_Dims_create(7, 2, dims)  ->  (7,1)
    call MPI_Dims_create(mpisize, 2, pdims, mpierror)

    ! determine my coordinates (x,y)
    px = pdims(1)
    py = pdims(2)
    rx = mod(mpirank, px)
    ry = mpirank / px

    ! determine my four neighbors
    north = (ry - 1) * px + rx; if( (ry - 1) < 0  ) north = MPI_PROC_NULL
    south = (ry + 1) * px + rx; if( (ry + 1) >= py) south = MPI_PROC_NULL
    west = ry * px + rx - 1;    if( (rx - 1) < 0  ) west  = MPI_PROC_NULL
    east = ry * px + rx + 1;    if( (rx + 1) >= px) east  = MPI_PROC_NULL

    ! decompose the domain
    bx = n / px             ! block size in x
    by = n / py             ! block size in y
    offx = (rx * bx) + 1    ! offset in x
    offy = (ry * by) + 1    ! offset in y

    ! initialize heat sources
    sources = reshape( [ n/2,   n/2,        &
                         n/3,   n/3,        &
                         n*4/5, n*8/9 ],    &
              shape(sources), order=[2, 1])

    do i = 1, nsources      ! determine which sources are in my patch
        locx = sources(i, 1) - offx
        locy = sources(i, 2) - offy
        if(locx >= 0 .and. locx <= bx .and. locy >= 0 .and. locy <= by) then
            locnsources = locnsources + 1
            locsources(locnsources, 1) = locx + 2
            locsources(locnsources, 2) = locy + 2
        endif
    enddo

    ! allocate communication buffers
```

```
  allocate(sendnorthgz(bx))   ! send buffers
  allocate(sendsouthgz(bx))
  allocate(recvnorthgz(bx))   ! receive buffers
  allocate(recvsouthgz(bx))
  ! allocate two work arrays
  allocate(aold(bx+2, by+2)); aold = 0.0   ! 1-wide halo zones!
  allocate(anew(bx+2, by+2)); anew = 0.0   ! 1-wide halo zones!

  do iters = 1, niters, 2

      ! --- anew <- stencil(aold) ---
      if(north /= MPI_PROC_NULL) then
          sendnorthgz = aold(2, 2:bx+1)
          recvnorthgz = 0.0
          call MPI_IRecv(recvnorthgz, bx, MPI_DOUBLE_PRECISION, north,  &
                          mpitag, MPI_COMM_WORLD, recvrequest(1), mpierror)
          call MPI_ISend(sendnorthgz, bx, MPI_DOUBLE_PRECISION, north,  &
                          mpitag, MPI_COMM_WORLD, sendrequest(1), mpierror)
      endif
      if(south /= MPI_PROC_NULL) then
          sendsouthgz = aold(bx+1, 2:bx+1)
          recvsouthgz(:) = 0.0
          call MPI_IRecv(recvsouthgz, bx, MPI_DOUBLE_PRECISION, south,  &
                          mpitag, MPI_COMM_WORLD, recvrequest(2), mpierror)
          call MPI_ISend(sendsouthgz, bx, MPI_DOUBLE_PRECISION, south,  &
                          mpitag, MPI_COMM_WORLD, sendrequest(2), mpierror)
      endif
      if(east /= MPI_PROC_NULL) then
          call MPI_IRecv(aold(2:bx+1, bx+2), bx, MPI_DOUBLE_PRECISION, east, &
                          mpitag, MPI_COMM_WORLD, recvrequest(3), mpierror)
          call MPI_ISend(aold(2:bx+1, bx+1), bx, MPI_DOUBLE_PRECISION, east, &
                          mpitag, MPI_COMM_WORLD, sendrequest(3), mpierror)
      endif
      if(west /= MPI_PROC_NULL) then
          call MPI_IRecv(aold(2:bx+1, 1), bx, MPI_DOUBLE_PRECISION, west, &
                         mpitag, MPI_COMM_WORLD, recvrequest(4), mpierror)
          call MPI_ISend(aold(2:bx+1, 2), bx, MPI_DOUBLE_PRECISION, west, &
                         mpitag, MPI_COMM_WORLD, sendrequest(4), mpierror)
          endif
      if(north /= MPI_PROC_NULL) then
          call MPI_Wait(recvrequest(1), MPI_STATUS_IGNORE, mpierror)
          call MPI_Wait(sendrequest(1), MPI_STATUS_IGNORE, mpierror)
          aold(1, 2:bx+1)=recvnorthgz
      endif
      if(south /= MPI_PROC_NULL) then
```

```
            call MPI_Wait(recvrequest(2), MPI_STATUS_IGNORE, mpierror)
            call MPI_Wait(sendrequest(2), MPI_STATUS_IGNORE, mpierror)
            aold(bx+2, 2:bx+1)=recvsouthgz
        endif
        if(east /= MPI_PROC_NULL) then
            call MPI_Wait(recvrequest(3), MPI_STATUS_IGNORE, mpierror)
            call MPI_Wait(sendrequest(3), MPI_STATUS_IGNORE, mpierror)
        endif
        if(west /= MPI_PROC_NULL) then
            call MPI_Wait(recvrequest(4), MPI_STATUS_IGNORE, mpierror)
            call MPI_Wait(sendrequest(4), MPI_STATUS_IGNORE, mpierror)
        endif

        ! update grid points
        do j = 2, by+1
            do i = 2, bx+1
                anew(i, j) = aold(i, j)/2.0 + (aold(i-1, j) + aold(i+1, j) +  &
                             aold(i, j-1) + aold(i, j+1)) / 4.0 / 2.0
            enddo
        enddo

        do i = 1, locnsources
            anew(locsources(i, 1), locsources(i, 2)) =   &
                anew(locsources(i, 1), locsources(i, 2)) + energy
        enddo

        ! --- aold <- stencil(anew) ---
        if(north /= MPI_PROC_NULL) then
            sendnorthgz=anew(2, 2:bx+1)
            call MPI_IRecv(recvnorthgz, bx, MPI_DOUBLE_PRECISION, north, mpitag,&
                                MPI_COMM_WORLD, recvrequest(1), mpierror)
            call MPI_ISend(sendnorthgz, bx, MPI_DOUBLE_PRECISION, north, mpitag,&
                                MPI_COMM_WORLD, sendrequest(1), mpierror)
        endif
        if(south /= MPI_PROC_NULL) then
            sendsouthgz=anew(bx+1, 2:bx+1)
            call MPI_IRecv(recvsouthgz, bx, MPI_DOUBLE_PRECISION, south, mpitag,&
                                MPI_COMM_WORLD, recvrequest(2), mpierror)
            call MPI_ISend(sendsouthgz, bx, MPI_DOUBLE_PRECISION, south, mpitag,&
                                MPI_COMM_WORLD, sendrequest(2), mpierror)
        endif
        if(east /= MPI_PROC_NULL) then
            call MPI_IRecv(anew(2:bx+1, bx+2), bx, MPI_DOUBLE_PRECISION, east,&
                                mpitag, MPI_COMM_WORLD, recvrequest(3), mpierror)
            call MPI_ISend(anew(2:bx+1, bx+1), bx, MPI_DOUBLE_PRECISION, east,&
                             mpitag, MPI_COMM_WORLD, sendrequest(3), mpierror)
        endif
        if(west /= MPI_PROC_NULL) then
            call MPI_IRecv(anew(2:bx+1, 1), bx, MPI_DOUBLE_PRECISION, west,
    mpitag,  &
                             MPI_COMM_WORLD, recvrequest(4), mpierror)
            call MPI_ISend(anew(2:bx+1, 2), bx, MPI_DOUBLE_PRECISION, west,
    mpitag,  &
                             MPI_COMM_WORLD, sendrequest(4), mpierror)
        endif
        if(north /= MPI_PROC_NULL) then
            call MPI_Wait(recvrequest(1), MPI_STATUS_IGNORE, mpierror)
            call MPI_Wait(sendrequest(1), MPI_STATUS_IGNORE, mpierror)
            anew(1, 2:bx+1)=recvnorthgz
        endif
        if(south /= MPI_PROC_NULL) then
            call MPI_Wait(recvrequest(2), MPI_STATUS_IGNORE, mpierror)
            call MPI_Wait(sendrequest(2), MPI_STATUS_IGNORE, mpierror)
            anew(bx+2, 2:bx+1)=recvsouthgz
        endif
        if(east /= MPI_PROC_NULL) then
            call MPI_Wait(recvrequest(3), MPI_STATUS_IGNORE, mpierror)
            call MPI_Wait(sendrequest(3), MPI_STATUS_IGNORE, mpierror)
        endif
        if(west /= MPI_PROC_NULL) then
            call MPI_Wait(recvrequest(4), MPI_STATUS_IGNORE, mpierror)
            call MPI_Wait(sendrequest(4), MPI_STATUS_IGNORE, mpierror)
        endif

        ! update grid points
        do j = 2, by+1
            do i = 2, bx+1
                aold(i, j) = anew(i, j)/2.0 + (anew(i-1, j) + anew(i+1, j) +  &
                             anew(i, j-1) + anew(i, j+1)) / 4.0 / 2.0
            enddo
        enddo

        do i = 1, locnsources
            aold(locsources(i, 1), locsources(i, 2)) =  &
                aold(locsources(i, 1), locsources(i, 2)) + energy
        enddo

    enddo

    ! ALL REDUCE:
```

```fortran
    heat = 0.0
    do j = 2, by+1
        do i = 2, bx+1
            heat = heat + aold(i, j)
        enddo
    enddo
    call MPI_Allreduce(heat, rheat, 1, MPI_DOUBLE_PRECISION, MPI_SUM,  &
                       MPI_COMM_WORLD, mpierror)

    orank = mpirank
    if(mpirank == 0) then
        otime = mpiwtime + MPI_Wtime()
        oheat = rheat
    endif

    call MPI_Finalize(mpierror)
end subroutine
```

Listing D.6 - Parallel F2PY implementation of the stencil test case - Python main code.

```python
from time import time
from stc_f2p_par import stm

n             = 4800    # nxn grid; 4800,1,500->1500; 100,1,10->30; [4800]
energy        = 1       # energy to be injected per iteration; [1]
niters        = 500     # number of iterations; [500]
heat          = 0.0
t             = 0.0
t0            = 0.0
rank          = 0

t0 = time()
heat, t, rank = stm(n, energy, niters)
t0 = time() - t0

if not rank :
    print("Heat = %0.4f | Time = %0.4f | TimePyt = %0.4f" %(heat, t, t0))
```

D.1.5 Serial Python

Listing D.7 - Serial Python implementation of the stencil test case.

```python
from time import time
t = time()
import numpy as np
```

```

n             = 4800    # nxn grid (4800,1,500)=1500
energy        = 1.0     # energy to be injected per iteration
niters        = 500     # number of iterations

size          = n + 2
sizeEnd       = n + 1
anew = aold   = np.zeros((size,  size), np.float64)
nsources      = 3       # sources of energy
sources       = np.empty((nsources, 2), np.int)
sources[:,:] = [ [n//2, n//2], [n//3, n//3], [n*4//5, n*8//9] ]
niters        = (niters+1) // 2

for iters in range(niters):
    anew[1:-1, 1:-1] =  ( aold[1:-1, 1:-1] / 2.0 +
                        ( aold[2:   , 1:-1] + aold[ :-2, 1:-1] +
                          aold[1:-1, 2:  ] + aold[1:-1,  :-2] ) / 8.0 )
    anew[sources[0:nsources,0], sources[0:nsources,1]] += energy
    aold[1:-1, 1:-1] =  ( anew[1:-1, 1:-1] / 2.0 +
                        ( anew[2:   , 1:-1] + anew[ :-2, 1:-1] +
                          anew[1:-1, 2:  ] + anew[1:-1,  :-2] ) / 8.0 )
    aold[sources[0:nsources,0], sources[0:nsources,1]] += energy
heat = np.sum( aold[1:sizeEnd, 1:sizeEnd] )  # system total heat

t = time() - t
print("Heat = %0.4f | Time = %0.4f s" %(heat, t))
```

D.1.6 Parallel Python

Listing D.8 - Parallel Python implementation of the stencil test case.

```
from mpi4py import MPI
import numpy as np

n             = 4800    # nxn grid (4800,1,500)=1500
energy        = 1.0     # energy to be injected per iteration
niters        = 500     # number of iterations

nsources      = 3       # sources of energy
sources       = np.zeros((nsources, 2), np.int)
sources[:,:] = [ [n//2, n//2], [n//3, n//3], [n*4//5, n*8//9] ]
locnsources   = locx = locy = 0     # number of sources in my area
locsources    = np.zeros((nsources, 2), np.int)  # local to my rank
rheat         = np.zeros(1, np.float64)
size          = n + 2
```

```
sizeEnd      = n + 1

comm = MPI.COMM_WORLD
mpirank = comm.rank
mpisize = comm.size
if not mpirank : mpiwtime = -MPI.Wtime()

# determine my coordinates (x,y)
pdims = MPI.Compute_dims(mpisize, 2)
px = pdims[0]
py = pdims[1]
rx = mpirank % px
ry = mpirank // px

# determine my four neighbors
north = (ry - 1) * px + rx
if (ry - 1) < 0 : north = MPI.PROC_NULL
south = (ry + 1) * px + rx
if (ry + 1) >= py : south = MPI.PROC_NULL
west = ry * px + rx - 1
if (rx - 1) < 0 : west = MPI.PROC_NULL
east = ry * px + rx + 1
if (rx + 1) >= px : east = MPI.PROC_NULL

# decompose the domain
bx = n // px           # block size in x
by = n // py           # block size in y
offx = rx * bx + 1     # offset in x
offy = ry * by + 1     # offset in y

# determine which sources are in my patch
for i in range(nsources) :
    locx = sources[i, 0] - offx
    locy = sources[i, 1] - offy
#    if(locx >= 0 and locx <= bx and locy >= 0 and locy <= by) :
    if(locx >= 0 and locx < bx and locy >= 0 and locy < by) :
        locsources[locnsources, 0] = locx + 2
        locsources[locnsources, 1] = locy + 2
        locnsources += 1

# working arrays with 1-wide halo zones
anew = np.zeros((bx+2, by+2), np.float64)
aold = np.zeros((bx+2, by+2), np.float64)

# iterations
```

```
niters = (niters + 1) // 2
for iters in range(niters) :
    # exchange data with neighbors
    if north != MPI.PROC_NULL :
        r1=comm.irecv(source=north, tag=1)
        s1=comm.isend(aold[1, 1:bx+1], dest=north, tag=1)
    if south != MPI.PROC_NULL :
        r2=comm.irecv(source=south, tag=1)
        s2=comm.isend(aold[bx, 1:bx+1], dest=south, tag=1)
    if east != MPI.PROC_NULL :
        r3 = comm.irecv(source=east, tag=1)
        s3 = comm.isend(aold[1:bx+1, bx], dest=east, tag=1)
    if west != MPI.PROC_NULL :
        r4 = comm.irecv(source=west, tag=1)
        s4 = comm.isend(aold[1:bx+1, 1], dest=west, tag=1)
    # wait
    if north != MPI.PROC_NULL :
        s1.wait()
        aold[0, 1:bx+1] = r1.wait()
    if south != MPI.PROC_NULL :
        s2.wait()
        aold[bx+1, 1:bx+1] = r2.wait()
    if east != MPI.PROC_NULL :
        s3.wait()
        aold[1:bx+1, bx+1] = r3.wait()
    if west != MPI.PROC_NULL :
        s4.wait
        aold[1:bx+1, 0] = r4.wait()
    # update grid
    anew[1:-1, 1:-1] =  ( aold[1:-1, 1:-1] / 2.0 +
                        ( aold[2:   , 1:-1] + aold[ :-2, 1:-1] +
                          aold[1:-1, 2:  ] + aold[1:-1,  :-2] ) / 8.0 )
    # refresh heat sources
    anew[locsources[0:locnsources, 0]-1, locsources[0:locnsources, 1]-1] +=
    energy

    # exchange data with neighbors
    if north != MPI.PROC_NULL :
        r1=comm.irecv(source=north, tag=1)
        s1=comm.isend(anew[1, 1:bx+1], dest=north, tag=1)
    if south != MPI.PROC_NULL :
        r2=comm.irecv(source=south, tag=1)
        s2=comm.isend(anew[bx, 1:bx+1], dest=south, tag=1)
    if east != MPI.PROC_NULL :
        r3 = comm.irecv(source=east, tag=1)
        s3 = comm.isend(anew[1:bx+1, bx], dest=east, tag=1)
    if west != MPI.PROC_NULL :
        r4 = comm.irecv(source=west, tag=1)
        s4 = comm.isend(anew[1:bx+1, 1], dest=west, tag=1)
    # wait
    if north != MPI.PROC_NULL :
        s1.wait()
        anew[0, 1:bx+1] = r1.wait()
    if south != MPI.PROC_NULL :
        s2.wait()
        anew[bx+1, 1:bx+1] = r2.wait()
    if east != MPI.PROC_NULL :
        s3.wait()
        anew[1:bx+1, bx+1] = r3.wait()
    if west != MPI.PROC_NULL :
        s4.wait
        anew[1:bx+1, 0] = r4.wait()
    # update grid
    aold[1:-1, 1:-1] =  ( anew[1:-1, 1:-1] / 2.0 +
                        ( anew[2:   , 1:-1] + anew[ :-2, 1:-1] +
                          anew[1:-1, 2:  ] + anew[1:-1,  :-2] ) / 8.0 )
    # refresh heat sources
    aold[locsources[0:locnsources, 0]-1, locsources[0:locnsources, 1]-1] +=
    energy

# get final heat in the system
comm.Reduce(np.sum(aold[1:bx+1, 1:by+1]), rheat)

# show
if not mpirank :
    print("Heat=%0.4f | Time=%0.4f | MPISize=%0s | Dim=%0s | bx,by=%0s,%0s"
          %(rheat, mpiwtime+MPI.Wtime(), mpisize, pdims, bx, by))
```

D.1.7 Serial Cython

Listing D.9 - Serial Cython implementation of the stencil test case - Cython module code.

```
#cython: boundscheck=False, wraparound=False, cdivision=True
#cython: initializedcheck=False, language_level=3, infer_types=True

cpdef st(int n, double energy, int niters):
    from time import time
    import numpy as np

    cdef double      heat      = 0.0
    cdef double      t         = 0.0
    cdef Py_ssize_t  size      = n + 2
    cdef Py_ssize_t  sizeStart = 1
    cdef Py_ssize_t  sizeEnd   = n + 1
    cdef Py_ssize_t  iters, i, j

    t = time()

    cdef double[:,::1] mvaold = np.zeros((size, size), np.double)
    cdef double[:,::1] mvanew = np.zeros((size, size), np.double)
    cdef Py_ssize_t    nsources  = 3      # qde de fontes
    cdef    int[:,::1] mvsources = np.empty( (nsources,2), np.intc)

    mvsources[0,0] = mvsources[0,1] = n/2
    mvsources[1,0] = mvsources[1,1] = n/3
    mvsources[2,0] = n*4/5
    mvsources[2,1] = n*8/9

    niters = (niters + 1) // 2
    for iters in range(niters) :
        # iteracao impar
        for i in range(sizeStart, sizeEnd) :
            for j in range(sizeStart, sizeEnd) :
                mvanew[i,j] = ( mvaold[i,j] / 2.0 +
                              ( mvaold[i-1,j] + mvaold[i+1,j] +
                                mvaold[i,j-1] + mvaold[i,j+1] ) / 8.0 )
        for i in range(nsources) :
            mvanew[mvsources[i,0], mvsources[i,1]] += energy
        # iteracao par
        for i in range(sizeStart, sizeEnd) :
            for j in range(sizeStart, sizeEnd) :
                mvaold[i,j] = ( mvanew[i,j] / 2.0 +
                              ( mvanew[i-1,j] + mvanew[i+1,j] +
                                mvanew[i,j-1] + mvanew[i,j+1] ) / 8.0 )
        for i in range(nsources) :
            mvaold[mvsources[i,0], mvsources[i,1]] += energy
    # calcula o total de energia
    for i in range(sizeStart, sizeEnd) :
        for j in range(sizeStart, sizeEnd) :
            heat += mvaold[i,j]
    t = time() - t
    return heat, t
```

Listing D.10 - Serial Cython implementation of the stencil test case - Python main code.

```
from time import time
tp = time()
import scs

n            = 4800   # nxn grid; 4800,1,500->1500; 100,1,10->30 [4800]
energy       = 1.0    # energy to be injected per iteration [1.0]
niters       = 500    # number of iterations [500]

heat, t = scs.st(n, energy, niters)
tp = time() - tp
print("Heat = %0.4f | Time = %0.4f | TimePyt = %0.4f" %(heat, t, tp))
```

D.1.8 Parallel Cython

Listing D.11 - Parallel Cython implementation of the stencil test case - Cython module.

```
#cython: language_level=3
#cython: cdivision=True
#cython: initializedcheck=False
#cython: infer_types=True
#cython: wraparound=False
#cython: boundscheck=False

import numpy as np

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] / 2.0 +
                         ( aold[i-1,j] + aold[i+1,j] +
                           aold[i,j-1] + aold[i,j+1] ) / 8.0 )
```

Listing D.12 - Parallel Cython implementation of the stencil test case - Python main code.

```
import numpy as np
import time
from mpi4py import MPI
import scp2

n            = 4800   # nxn grid (4800,1,500)=1500; (100,1,10)=30
energy       = 1.0    # energy to be injected per iteration
niters       = 500    # number of iterations

nsources     = 3      # sources of energy
size         = n + 2
```

```
heat         = np.zeros((1), np.float64)     # system total heat
anew         = np.zeros((size, size), np.float64)
aold         = np.zeros((size, size), np.float64)
sources      = np.empty((3,2), np.int32)
sources[:,:] = [ [n//2, n//2], [n//3, n//3], [n*4//5, n*8//9] ]
niters       = (niters+1) // 2

comm = MPI.COMM_WORLD
mpirank = comm.rank
mpisize = comm.size

nsources = 3
sources = np.zeros((nsources, 2), np.intc)
sources[:,:] = [ [n//2, n//2], [n//3, n//3], [n*4//5, n*8//9] ]

# sources in my area, local to my rank
locnsources = 0
locsources = np.empty((nsources,2), np.intc)

rheat = np.zeros(1, np.double)
bheat = np.zeros(1, np.double)

# determine my coordinates (x,y)
pdims = MPI.Compute_dims(mpisize, 2)
px    = pdims[0]
py    = pdims[1]
rx    = mpirank % px
ry    = mpirank // px

# determine my four neighbors
north = (ry - 1) * px + rx
if (ry - 1) < 0 :
    north = MPI.PROC_NULL
south = (ry + 1) * px + rx
if (ry + 1) >= py :
    south = MPI.PROC_NULL
west = ry * px + rx - 1
if (rx - 1) < 0 :
    west = MPI.PROC_NULL
east = ry * px + rx + 1
if (rx + 1) >= px :
    east = MPI.PROC_NULL

# decompose the domain
bx = n // px            # block size in x
```

```
by = n // py            # block size in y
offx = rx * bx + 1      # offset in x
offy = ry * by + 1      # offset in y

# determine which sources are in my patch
for i in range(nsources) :
    locx = sources[i, 0] - offx
    locy = sources[i, 1] - offy
    if(locx >= 0 and locx <= bx and locy >= 0 and locy <= by) :
        locsources[locnsources, 0] = locx + 2 - 1
        locsources[locnsources, 1] = locy + 2 - 1
        locnsources += 1

# working arrays with 1-wide halo zones
anew = np.zeros((bx+2, by+2), np.double)
aold = np.zeros((bx+2, by+2), np.double)

if not mpirank : t0 = time.time()

for iters in range(niters) :
    # exchange data with neighbors
    if north != MPI.PROC_NULL :
        r1=comm.irecv(source=north, tag=1)
        s1=comm.isend(aold[1, 1:bx+1], dest=north, tag=1)
    if south != MPI.PROC_NULL :
        r2=comm.irecv(source=south, tag=1)
        s2=comm.isend(aold[bx, 1:bx+1], dest=south, tag=1)
    if east != MPI.PROC_NULL :
        r3 = comm.irecv(source=east, tag=1)
        s3 = comm.isend(aold[1:bx+1, bx], dest=east, tag=1)
    if west != MPI.PROC_NULL :
        r4 = comm.irecv(source=west, tag=1)
        s4 = comm.isend(aold[1:bx+1, 1], dest=west, tag=1)
    # wait
    if north != MPI.PROC_NULL :
        s1.wait()
        aold[0, 1:bx+1] = r1.wait()
    if south != MPI.PROC_NULL :
        s2.wait()
        aold[bx+1, 1:bx+1] = r2.wait()
    if east != MPI.PROC_NULL :
        s3.wait()
        aold[1:bx+1, bx+1] = r3.wait()
    if west != MPI.PROC_NULL :
        s4.wait
        aold[1:bx+1, 0] = r4.wait()

    # update grid
    scp2.stp(anew, aold, bx, by)

    # refresh heat sources
    for i in range(locnsources) :
        anew[locsources[i, 0]-1, locsources[i, 1]-1] += energy

    # exchange data with neighbors
    if north != MPI.PROC_NULL :
        r1=comm.irecv(source=north, tag=1)
        s1=comm.isend(anew[1, 1:bx+1], dest=north, tag=1)
    if south != MPI.PROC_NULL :
        r2=comm.irecv(source=south, tag=1)
        s2=comm.isend(anew[bx, 1:bx+1], dest=south, tag=1)
    if east != MPI.PROC_NULL :
        r3 = comm.irecv(source=east, tag=1)
        s3 = comm.isend(anew[1:bx+1, bx], dest=east, tag=1)
    if west != MPI.PROC_NULL :
        r4 = comm.irecv(source=west, tag=1)
        s4 = comm.isend(anew[1:bx+1, 1], dest=west, tag=1)
    # wait
    if north != MPI.PROC_NULL :
        s1.wait()
        anew[0, 1:bx+1] = r1.wait()
    if south != MPI.PROC_NULL :
        s2.wait()
        anew[bx+1, 1:bx+1] = r2.wait()
    if east != MPI.PROC_NULL :
        s3.wait()
        anew[1:bx+1, bx+1] = r3.wait()
    if west != MPI.PROC_NULL :
        s4.wait
        anew[1:bx+1, 0] = r4.wait()

    # update grid
    scp2.stp(aold, anew, bx, by)

    # refresh heat sources
    for i in range(locnsources) :
        aold[locsources[i, 0]-1, locsources[i, 1]-1] += energy

# get final heat in the system
bheat[0] = np.sum(aold[1:-1, 1:-1])
```

```
comm.Reduce(bheat, rheat)

if not mpirank :
    t1 = MPI.Wtime() - t0
    print('Heat={:0.4f} | Time={:0.4f} | MPISize={:d} | Dim={:d},{:d} | bx,by={:d
    },{:d}'
          .format(rheat[0], t1, mpisize, pdims[0], pdims[1], bx, by))
```

D.1.9 Serial Numba-CPU

Listing D.13 - Serial Numba-CPU implementation of the stencil test case.

```
import numpy as np
from numba import jit, config, prange
from time import time

config.DUMP_ASSEMBLY = 0
config.NUMBA_ENABLE_AVX = 1
config.NUMBA_NUM_THREADS = 1

@jit('(float64[:,:],float64[:,:])', nopython=True, parallel=True, nogil=True)
def kernel_seq(anew, aold) :
    anew[1:-1, 1:-1] = ( aold[1:-1, 1:-1] * 0.5 +
                       ( aold[2:   , 1:-1] + aold[ :-2, 1:-1] +
                         aold[1:-1, 2:   ] + aold[1:-1,  :-2] ) * 0.125 )

n             = 4800    # nxn grid (4800,1,500)=1500; (4800,1,5)=12
energy        = 1.0     # energy to be injected per iteration
niters        = 500     # number of iterations
nsources      = 3       # sources of energy
size          = n + 2
sizeEnd       = n + 1
heat          = np.zeros((1), np.float64)     # system total heat
anew          = np.zeros((size,  size), np.float64)
aold          = np.zeros((size,  size), np.float64)
sources       = np.empty((nsources, 2), np.int16)
sources[:,:] = [ [n//2, n//2], [n//3, n//3], [n*4//5, n*8//9] ]
niters        = (niters + 1) // 2

t0 = time()
for iters in range(niters) :
    kernel_seq(anew, aold)
    for i in range(nsources) :
        anew[sources[i, 0], sources[i, 1]] += energy
    kernel_seq(aold, anew)
```

```python
    for i in range(nsources) :
        aold[sources[i, 0], sources[i, 1]] += energy

heat[0] = np.sum( aold[1:-1, 1:-1] )  # system total heat
t0 = time() - t0

print("Heat = %0.4f | Time = %0.4f | Thread count = %s" %
      (heat[0], t0, config.NUMBA_NUM_THREADS))
```

D.1.10 Parallel Numba-CPU

Listing D.14 - Parallel Numba-CPU implementation of the stencil test case.

```python
import numpy as np
import time
from mpi4py import MPI

from numba import jit, prange, config
config.DUMP_ASSEMBLY = 0
config.NUMBA_ENABLE_AVX = 1
config.NUMBA_NUM_THREADS = 1

@jit('(float64[:,:],float64[:,:])', nopython=True, parallel=True, nogil=True)
def kernel1(anew, aold) :
    anew[1:-1, 1:-1] = ( aold[1:-1, 1:-1] * 0.5 +
                       ( aold[2:  , 1:-1] + aold[ :-2, 1:-1] +
                         aold[1:-1, 2:  ] + aold[1:-1,  :-2] ) * 0.125 )

n             = 4800    # nxn grid (4800,1,500)=1500; (100,1,10)=30
energy        = 1.0     # energy to be injected per iteration
niters        = 500     # number of iterations

nsources      = 3       # sources of energy
size          = n + 2
heat          = np.zeros((1), np.float64)     # system total heat
anew          = np.zeros((size, size), np.float64)
aold          = np.zeros((size, size), np.float64)
sources       = np.empty((3,2), np.int32)
sources[:,:] = [ [n//2, n//2], [n//3, n//3], [n*4//5, n*8//9] ]
niters        = (niters+1) // 2

comm = MPI.COMM_WORLD
mpirank = comm.rank
mpisize = comm.size

```

```
nsources = 3
sources = np.zeros((nsources, 2), np.intc)
sources[:,:] = [ [n//2, n//2], [n//3, n//3], [n*4//5, n*8//9] ]

# sources in my area, local to my rank
locnsources = 0
locsources = np.empty((nsources,2), np.intc)

rheat = np.zeros(1, np.double)
bheat = np.zeros(1, np.double)

# determine my coordinates (x,y)
pdims = MPI.Compute_dims(mpisize, 2)
px    = pdims[0]
py    = pdims[1]
rx    = mpirank % px
ry    = mpirank // px

# determine my four neighbors
north = (ry - 1) * px + rx
if (ry - 1) < 0 :
    north = MPI.PROC_NULL
south = (ry + 1) * px + rx
if (ry + 1) >= py :
    south = MPI.PROC_NULL
west = ry * px + rx - 1
if (rx - 1) < 0 :
    west = MPI.PROC_NULL
east = ry * px + rx + 1
if (rx + 1) >= px :
    east = MPI.PROC_NULL

# decompose the domain
bx = n // px            # block size in x
by = n // py            # block size in y
offx = rx * bx + 1      # offset in x
offy = ry * by + 1      # offset in y

# determine which sources are in my patch
for i in range(nsources) :
    locx = sources[i, 0] - offx
    locy = sources[i, 1] - offy
    if(locx >= 0 and locx <= bx and locy >= 0 and locy <= by) :
        locsources[locnsources, 0] = locx + 2 - 1
        locsources[locnsources, 1] = locy + 2 - 1
        locnsources += 1

# working arrays with 1-wide halo zones
anew = np.zeros((bx+2, by+2), np.double)
aold = np.zeros((bx+2, by+2), np.double)

if not mpirank : t0 = time.time()

for iters in range(niters) :
    # exchange data with neighbors
    if north != MPI.PROC_NULL :
        r1=comm.irecv(source=north, tag=1)
        s1=comm.isend(aold[1, 1:bx+1], dest=north, tag=1)
    if south != MPI.PROC_NULL :
        r2=comm.irecv(source=south, tag=1)
        s2=comm.isend(aold[bx, 1:bx+1], dest=south, tag=1)
    if east != MPI.PROC_NULL :
        r3 = comm.irecv(source=east, tag=1)
        s3 = comm.isend(aold[1:bx+1, bx], dest=east, tag=1)
    if west != MPI.PROC_NULL :
        r4 = comm.irecv(source=west, tag=1)
        s4 = comm.isend(aold[1:bx+1, 1], dest=west, tag=1)
    # wait
    if north != MPI.PROC_NULL :
        s1.wait()
        aold[0, 1:bx+1] = r1.wait()
    if south != MPI.PROC_NULL :
        s2.wait()
        aold[bx+1, 1:bx+1] = r2.wait()
    if east != MPI.PROC_NULL :
        s3.wait()
        aold[1:bx+1, bx+1] = r3.wait()
    if west != MPI.PROC_NULL :
        s4.wait
        aold[1:bx+1, 0] = r4.wait()

    # update grid
    kernel1(anew, aold)

    # refresh heat sources
    for i in range(locnsources) :
        anew[locsources[i, 0]-1, locsources[i, 1]-1] += energy

    # exchange data with neighbors
    if north != MPI.PROC_NULL :
        r1=comm.irecv(source=north, tag=1)
        s1=comm.isend(anew[1, 1:bx+1], dest=north, tag=1)
    if south != MPI.PROC_NULL :
        r2=comm.irecv(source=south, tag=1)
        s2=comm.isend(anew[bx, 1:bx+1], dest=south, tag=1)
    if east != MPI.PROC_NULL :
        r3 = comm.irecv(source=east, tag=1)
        s3 = comm.isend(anew[1:bx+1, bx], dest=east, tag=1)
    if west != MPI.PROC_NULL :
        r4 = comm.irecv(source=west, tag=1)
        s4 = comm.isend(anew[1:bx+1, 1], dest=west, tag=1)
    # wait
    if north != MPI.PROC_NULL :
        s1.wait()
        anew[0, 1:bx+1] = r1.wait()
    if south != MPI.PROC_NULL :
        s2.wait()
        anew[bx+1, 1:bx+1] = r2.wait()
    if east != MPI.PROC_NULL :
        s3.wait()
        anew[1:bx+1, bx+1] = r3.wait()
    if west != MPI.PROC_NULL :
        s4.wait
        anew[1:bx+1, 0] = r4.wait()

    # update grid
    kernel1(aold, anew)

    # refresh heat sources
    for i in range(locnsources) :
        aold[locsources[i, 0]-1, locsources[i, 1]-1] += energy

# get final heat in the system
bheat[0] = np.sum(aold[1:-1, 1:-1])
comm.Reduce(bheat, rheat)

if not mpirank :
    t1 = MPI.Wtime() - t0
    print('Heat={:0.4f} | Time={:0.4f} | MPISize={:d} | Dim={:d},{:d} | bx,by={:d},{:d}'
          .format(rheat[0], t1, mpisize, pdims[0], pdims[1], bx, by))
```

D.1.11 Numba-GPU

Listing D.15 - Numba-GPU implementation of the stencil test case.

```python
import math
from time import time
import numpy as np
from numba import cuda, jit, prange

@cuda.jit
def st3(a1, a2):
    n = a1.shape[0] - 1
    i, j = cuda.grid(2)
    if (i > 0 and j > 0) and (i < n and j < n) :
        a1[i,j] = a2[i,j]/2.0+(a2[i-1,j]+a2[i+1,j]+a2[i,j-1]+a2[i,j+1])/8.0

def calc3(anew, aold, heat, sizeEnd, niters, nsources, sources, energy,
          blocks_per_grid, threads_per_block):
    for iters in range(0, niters, 2):
        st3[blocks_per_grid, threads_per_block](anew, aold)
        for i in range(0, nsources) :
            anew[sources[i,0], sources[i,1]] += energy    # heat source
        st3[blocks_per_grid, threads_per_block](aold, anew)
        for i in range(0, nsources):
            aold[sources[i,0], sources[i,1]] += energy    # heat source

#def par_cuda():
n             = 4800    # nxn grid
energy        = 1       # energy to be injected per iteration
niters        = 500     # number of iterations
nsources      = 3       # sources of energy
size          = n + 2   # plus the ghost zone
sizeEnd       = n + 1

# initialize the data arrays
anew          = np.zeros((size, size), np.float64)
aold          = np.zeros((size, size), np.float64)
# initialize three heat sources
sources       = np.empty((3,2), np.int32)
sources[:,:] = [ [n//2, n//2], [n//3, n//3], [n*4//5, n*8//9] ]
heat          = 0       # system total heat sum

# copy the arrays to the device
anew_global_mem = cuda.to_device(anew)
aold_global_mem = cuda.to_device(aold)

# configure blocks & grids
# set the number of threads in a block
threads_per_block = (32, 32)
# calculate the number of thread blocks in the grid
blocks_per_grid_x = math.ceil(aold.shape[0] / threads_per_block[0])
blocks_per_grid_y = math.ceil(aold.shape[1] / threads_per_block[1])
blocks_per_grid   = (blocks_per_grid_x, blocks_per_grid_y)

t = time()
# main calc
calc3(anew_global_mem, aold_global_mem, heat,
    sizeEnd, niters, nsources, sources, energy,
    blocks_per_grid, threads_per_block)

# copy the result back to the host
aold = aold_global_mem.copy_to_host()

for j in range(1, sizeEnd):
    for i in range(1, sizeEnd):
        heat = heat + aold[i,j]
t = time() - t

# show the result if desired
print("Heat=%.4f | Time=%.4f" % (heat, t))
```

D.2 Implementations of the FFT test case

D.2.1 Serial F90

Listing D.16 - Serial F90 implementation of the FFT test case.

```fortran
program main
    use, intrinsic :: iso_c_binding
    implicit none
    include "fftw3.f03"
    integer, parameter :: L = 576, M = 576, N = 576
    type(C_PTR) :: plan, cdata
    complex(C_DOUBLE_COMPLEX), pointer :: data(:,:,:)
    complex(C_DOUBLE_COMPLEX) :: s
    integer :: i, j, k
    double precision :: t0, t1, t2

    call cpu_time(t0)    ! time measurement

    ! in-place transform (note dimension reversal)
    cdata = fftw_alloc_complex(int(L * M * N, C_SIZE_T))
    call c_f_pointer(cdata, data, [L, M, N])

    ! create plan for in-place forward DFT (note dimension reversal)
    plan = fftw_plan_dft_3d(N, M, L, data, data, &
                            FFTW_FORWARD, FFTW_ESTIMATE)

    ! fills the array with complex values
    do k = 1, N
        do j = 1, M
            do i = 1, L
                data(i, j, k) = sin( real(i + j + k) )
            enddo
        enddo
    enddo
    data = dcmplx( real(data) , 0 )

    call cpu_time(t1)    ! time measurement

    ! compute transform (as many times as desired)
    call fftw_execute_dft(plan, data, data)
    ! checksum
    s = sum(data)

    call cpu_time(t2)    ! time measurement

    call fftw_destroy_plan(plan)
    call fftw_free(cdata)

    ! result
    write(*, "('S: 'spf0.0spf0.0'j')", advance="no") s * 1e-5
    write(*, "(' | L: 'g0)", advance="no") L
    write(*, "(' | T1: 'sf0.4)", advance="no") t1-t0
    write(*, "(' | TF: 'sf0.4)", advance="no") t2-t1
    write(*, "(' | TT: 'sf0.4)") t2-t0

end
```

D.2.2 Parallel F90

Listing D.17 - Parallel F90 implementation of the FFT test case.

```fortran
program main
    use, intrinsic :: iso_c_binding
    use MPI
    implicit none
```

```fortran
include 'fftw3-mpi.f03'
integer :: mpirank, mpisize, mpierror, i, j, k
integer(C_INTPTR_T), parameter :: L = 576, M = 576, N = 576
type(C_PTR) :: plan, cdata
complex(C_DOUBLE_COMPLEX), pointer :: data(:,:,:)
integer(C_INTPTR_T) :: alloc_local, local_N, local_start
complex(C_DOUBLE_COMPLEX) :: s, rs
double precision :: t0, t1, t2

call cpu_time(t0)    ! time measurement

call MPI_Init(mpierror)
call MPI_Comm_rank(MPI_COMM_WORLD, mpirank, mpierror)
call MPI_Comm_size(MPI_COMM_WORLD, mpisize, mpierror)

! init
call fftw_mpi_init()

! get local data size and allocate (note dimension reversal)
alloc_local = fftw_mpi_local_size_3d(N, M, L,  &
             MPI_COMM_WORLD, local_N, local_start)
cdata = fftw_alloc_complex(alloc_local)
call c_f_pointer(cdata, data, [L, M, local_N])

! create MPI plan for in-place forward DFT (note dimension reversal)
plan = fftw_mpi_plan_dft_3d(N, M, L, data, data,  &
            MPI_COMM_WORLD, FFTW_FORWARD, FFTW_ESTIMATE)

! Fills the array with complex values
do k = 1, int(local_N)
    do j = 1, M
        do i = 1, L
            data(i, j, k) = dcmplx( sin( real(i + j + (k + local_start)) ) ,
0)
        enddo
    enddo
enddo

call cpu_time(t1)    ! time measurement

! Compute transform (as many times as desired)
call fftw_mpi_execute_dft(plan, data, data)

! Checksum
s = sum(data)
```

```fortran
    call MPI_Reduce(s,                    &! send data
                    rs,                   &! recv data
                    1,                    &! count
                    MPI_DOUBLE_COMPLEX,   &! data type
                    MPI_SUM,              &! operation
                    0,                    &! rank of root process
                    MPI_COMM_WORLD, mpierror)

    ! clean

    call cpu_time(t2)    ! time measurement

    call fftw_destroy_plan(plan)
    call fftw_free(cdata)
    call fftw_mpi_cleanup()
    call mpi_finalize(mpierror)

    ! show the result
    if (mpirank == 0) then
        write(*, "('S: 'spf0.0spf0.0'j')", advance="no") rs * 1e-5
        write(*, "(' | L: 'g0)", advance="no") L
        write(*, "(' | N: 'g0)", advance="no") mpisize
        write(*, "(' | T1: 'sf0.4)", advance="no") t1-t0
        write(*, "(' | TF: 'sf0.4)", advance="no") t2-t1
        write(*, "(' | TT: 'sf0.4)") t2-t0
    endif

end
```

D.2.3 Serial F2PY

Listing D.18 - Serial F2PY implementation of the FFT test case - F90 module code.

```fortran
subroutine fs(ss, ll, ts, tf, tt)
    use, intrinsic :: iso_c_binding
    include "fftw3.f03"
    double complex, intent(out) :: ss
    integer, intent(out) :: ll
    double precision, intent(out) :: ts, tf, tt
    integer, parameter :: L = 576, M = 576, N = 576
    type(C_PTR) :: plan, cdata
    complex(C_DOUBLE_COMPLEX), pointer :: data(:,:,:)
    complex(C_DOUBLE_COMPLEX) :: s
    integer :: i, j, k
    double precision :: t0, t1, t2

    call cpu_time(t0)    ! time measurement

    ! in-place transform (note dimension reversal)
    cdata = fftw_alloc_complex(int(L * M * N, C_SIZE_T))
    call c_f_pointer(cdata, data, [L, M, N])

    ! create plan for in-place forward DFT (note dimension reversal)
    plan = fftw_plan_dft_3d(N, M, L, data, data, &
                            FFTW_FORWARD, FFTW_ESTIMATE)

    ! fills the array with complex values
    do k = 1, N
        do j = 1, M
            do i = 1, L
                data(i, j, k) = sin( real(i + j + k) )
            enddo
        enddo
    enddo
    data = dcmplx( real(data), 0 )

    call cpu_time(t1)    ! time measurement

    ! compute transform (as many times as desired)
    call fftw_execute_dft(plan, data, data)
    ! checksum
    s = sum(data)

    call cpu_time(t2)    ! time measurement

    call fftw_destroy_plan(plan)
    call fftw_free(cdata)

    ! result
    ss = s * 1e-5
    ll = L
    ts = t1 - t0
    tf = t2 - t1
    tt = t2 - t0

end subroutine
```

Listing D.19 - Serial F2PY implementation of the FFT test case - Python main code.

```python
import nc2cs
import time as tm
t2 = tm.time()     # time measurement
S, L, ts, tf, tt = nc2cs.fs()
t3 = tm.time()     # time measurement
print(f"S:{S:.0f}", end='')
print(f" | L:{L:0g}", end='')
print(f" | T1:{ts:.4f}", end='')
print(f" | TF:{tf:.4f}", end='')
print(f" | TT:{tt:.4f}", end='')
print(f" | TO:{t3-t2:.4f}")
```

D.2.4 Parallel F2PY

Listing D.20 - Parallel F2PY implementation of the FFT test case - F90 module code.

```fortran
subroutine fs(ss, ll, ts, tf, tt, mr, ms)
    use, intrinsic :: iso_c_binding
    use MPI
    implicit none
    include 'fftw3-mpi.f03'
    integer, intent(out) :: mr, ms, ll
    double complex, intent(out) :: ss
    double precision, intent(out) :: ts, tf, tt
    integer :: mpirank, mpisize, mpierror, i, j, k
    integer(C_INTPTR_T), parameter :: L = 576, M = 576, N = 576
    type(C_PTR) :: plan, cdata
    complex(C_DOUBLE_COMPLEX), pointer :: data(:,:,:)
    integer(C_INTPTR_T) :: alloc_local, local_N, local_start
    complex(C_DOUBLE_COMPLEX) :: s, rs
    real(C_DOUBLE) :: t0, t1, t2

    call cpu_time(t0)    ! time measurement

    call MPI_Init(mpierror)
    call MPI_Comm_rank(MPI_COMM_WORLD, mpirank, mpierror)
    call MPI_Comm_size(MPI_COMM_WORLD, mpisize, mpierror)

    ! init
    call fftw_mpi_init()

    ! get local data size and allocate (note dimension reversal)
    alloc_local = fftw_mpi_local_size_3d(N, M, L,  &
                 MPI_COMM_WORLD, local_N, local_start)
    cdata = fftw_alloc_complex(alloc_local)
    call c_f_pointer(cdata, data, [L, M, local_N])
```

```fortran

  ! create MPI plan for in-place forward DFT (note dimension reversal)
  plan = fftw_mpi_plan_dft_3d(N, M, L, data, data,  &
              MPI_COMM_WORLD, FFTW_FORWARD, FFTW_ESTIMATE)

  ! fill array with complex values
  do k = 1, int(local_N)
      do j = 1, M
          do i = 1, L
              data(i, j, k) = sin( real(i + j + (k + local_start)) )
          enddo
      enddo
  enddo
  data = dcmplx( real(data), 0 )

  call cpu_time(t1)    ! time measurement

  ! compute transform (as many times as desired)
  call fftw_mpi_execute_dft(plan, data, data)

  ! compute the checksum of processes
  s = sum(data)
  call MPI_Reduce(s,                    &! send data
                  rs,                   &! recv data
                  1,                    &! count
                  MPI_DOUBLE_COMPLEX,   &! data type
                  MPI_SUM,              &! operation
                  0,                    &! rank of root process
                  MPI_COMM_WORLD, mpierror)

  ! clean
  call fftw_destroy_plan(plan)
  call fftw_free(cdata)
  call fftw_mpi_cleanup()
  call mpi_finalize(mpierror)

  call cpu_time(t2)    ! time measurement

  ! result
  ss = rs * 1e-5
  ll = L
  ts = t1 - t0
  tf = t2 - t1
  tt = t2 - t0
  mr = mpirank
```

```
    ms = mpisize

end subroutine
```

Listing D.21 - Parallel F2PY implementation of the FFT test case - Python main code.

```python
import nc2cp
import time as tm
t2 = tm.time()    # time measurement
ss, ll, ts, tf, tt, mr, ms = nc2cp.fs()
t3 = tm.time()    # time measurement
if mr == 0 :
    print(f"S:{ss:.0f}", end='')
    print(f", L:{ll:0g}", end='')
    print(f", N:{ms:0g}", end='')
    print(f", T1:{ts:.4f}", end='')
    print(f", TF:{tf:.4f}", end='')
    print(f", TT:{tt:.4f}", end='')
    print(f", TO:{t3-t2:.4f}")
```

D.2.5 Serial Python

Listing D.22 - Serial Python implementation of the FFT test case.

```python
import numpy as np, pyfftw as pf, time as tm

t0 = tm.time()    # time measurement

# data
L = M = N = 576
u = pf.empty_aligned( (N, M, L), dtype=np.complex128 )
for k in range (u.shape[2]) :
    for j in range(u.shape[1]) :
        for i in range(u.shape[0]) :
            u[i, j, k] = i + j + k + 3
u.real = np.sin ( u.real )
u.imag = 0

t1 = tm.time()    # time measurement

# FFT
uf = pf.interfaces.numpy_fft.fftn(u,
        overwrite_input=True, auto_contiguous=False,
        auto_align_input=False)
# checksum
```

```python
S = np.sum(uf)

t2 = tm.time()    # time measurement

print(f"S: {S*1E-5:.0f}", end='')
print(f" | L: {L:0g}", end='')
print(f" | T1: {t1-t0:.4f}", end='')
print(f" | TF: {t2-t1:.4f}", end='')
print(f" | TT: {t2-t0:.4f}")
```

D.2.6 Parallel Python

Listing D.23 - Parallel Python implementation of the FFT test case.

```python
import numpy as np, time as tm
from mpi4py_fft import PFFT, newDistArray
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

t0 = tm.time()    # time measurement

# data
L = M = N = 576
NA = np.array([N, M, L], dtype=int)
f = PFFT(comm, NA, dtype=np.complex128, backend='pyfftw')
u = newDistArray(f, False)
for k in range (u.shape[2]) :
    for j in range(u.shape[1]) :
        for i in range(u.shape[0]) :
            u[i, j, k] = i + j + k + 3
u.real = np.sin ( u.real )
u.imag = 0

t1 = tm.time()    # time measurement

# FFT
uf = f.forward(u, normalize=False)

# checksum
S  = np.array(0, dtype=np.complex128)
Sn = np.array(np.sum(uf), dtype=np.complex128)
comm.Reduce([Sn, MPI.DOUBLE_COMPLEX], [S, MPI.DOUBLE_COMPLEX],
            op=MPI.SUM, root=0)
```

```python

t2 = tm.time()    # time measurement

if rank == 0 :
    print(f"S: {S*1E-5:.0f}", end='')
    print(f" | L: {L:0g}", end='')
    print(f" | N: {size:0g}", end='')
    print(f" | TS: {t1-t0:.4f}", end='')
    print(f" | TP: {t2-t1:.4f}", end='')
    print(f" | TT: {t2-t0:.4f}")
```

D.2.7 Serial Cython

Listing D.24 - Serial Cython implementation of the FFT test case - Cython module.

```python
#cython: boundscheck=False, wraparound=False, cdivision=True
#cython: initializedcheck=False, language_level=3, infer_types=True
def ff():
    import numpy as np, pyfftw as pf, time as tm

    t0 = tm.time()    # time measurement

    # data
    L = M = N = 576
    u = pf.empty_aligned( (N, M, L), dtype=np.complex128 )
    for k in range (u.shape[2]) :
        for j in range(u.shape[1]) :
            for i in range(u.shape[0]) :
                u[i, j, k] = i + j + k + 3
    u.real = np.sin ( u.real )
    u.imag = 0

    t1 = tm.time()    # time measurement

    # FFT
    uf = pf.interfaces.numpy_fft.fftn(u,
            overwrite_input=True, auto_contiguous=False,
            auto_align_input=False)
    # checksum
    s = np.sum(uf)

    t2 = tm.time()    # time measurement

    return s, L, t0, t1, t2
```

Listing D.25 - Serial Cython implementation of the FFT test case - Python main code.

```python
import numpy as np
import time as tm
import cc2cs

t3 = tm.time()    # time measurement

s, L, t0, t1, t2 = cc2cs.ff()

t4 = tm.time()    # time measurement

print(f"S:{s*1E-5:.0f}", end='')
print(f" | L:{L:0g}", end='')
print(f" | T1:{t1-t0:.4f}", end='')
print(f" | TF:{t2-t1:.4f}", end='')
print(f" | TT:{t2-t0:.4f}", end='')
print(f" | TO:{t4-t3:.4f}")
```

D.2.8 Parallel Cython

Listing D.26 - Parallel implementation of the FFT test case - Cython module.

```python
#cython: boundscheck=False, wraparound=False, cdivision=True
#cython: initializedcheck=False, language_level=3, infer_types=True
import numpy as np, time as tm
from mpi4py_fft import PFFT, newDistArray
from mpi4py import MPI

def ffp():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    t0 = tm.time()    # time measurement

    # data
    L = M = N = 576
    NA = np.array([N, M, L], dtype=int)
    f = PFFT(comm, NA, dtype=np.complex128, backend='pyfftw')
    u = newDistArray(f, False)
    for k in range (u.shape[2]) :
        for j in range(u.shape[1]) :
            for i in range(u.shape[0]) :
                u[i, j, k] = i + j + k + 3
    u.real = np.sin ( u.real )
    u.imag = 0

    t1 = tm.time()    # time measurement

    # FFT
    u_hat = f.forward(u, normalize=False)
    # checksum
    rs = np.array(0, dtype=np.complex128)
    s = np.array(np.sum(u_hat), dtype=np.complex128)
    comm.Reduce([s, MPI.DOUBLE_COMPLEX], [rs, MPI.DOUBLE_COMPLEX],
                op=MPI.SUM, root=0)

    t2 = tm.time()    # time measurement

    return rs, L, size, rank, t0, t1, t2
```

Listing D.27 - Parallel implementation of the FFT test case - Python main code.

```python
import numpy as np
import time as tm
import cc2cp

t3 = tm.time()    # time measurement

s, l, n, r, t0, t1, t2 = cc2cp.ffp()

t4 = tm.time()    # time measurement

if r == 0 :
    print(f"S:{s*1E-5:.0f}", end='')
    print(f", L:{l:0g}", end='')
    print(f", N:{n:0g}", end='')
    print(f", T1:{t1-t0:.4f}", end='')
    print(f", TF:{t2-t1:.4f}", end='')
    print(f", TT:{t2-t0:.4f}", end='')
    print(f", TO:{t4-t3:.4f}")
```

D.2.9 Serial Numba-CPU

Listing D.28 - Serial Numba-CPU implementation of the FFT test case.

```python
import numpy as np, pyfftw as pf, time as tm
from numba import njit, objmode

```

```python
t3 = tm.time()    # time measurement

@njit
def ff() :
    with objmode(t0 = 'f8') :
        t0 = tm.time()    # time measurement

    # data
    L = M = N = 576
    with objmode(u = 'complex128[:,:,:]') :  # annotate return type
        u = pf.empty_aligned( (N, M, L), dtype=np.complex128 )
    for k in range (u.shape[2]) :
        for j in range(u.shape[1]) :
            for i in range(u.shape[0]) :
                u[i, j, k] = complex( np.sin ( i + j + k + 3 ), 0 )

    with objmode(t1 = 'f8') :
        t1 = tm.time()    # time measurement

    # FFT
    with objmode(u = 'complex128[:,:,:]') :  # annotate return type
        u = pf.interfaces.numpy_fft.fftn(u)
    # checksum
    s = np.sum(u)

    with objmode(t2 = 'f8') :
        t2 = tm.time()    # time measurement

    return s, L, t0, t1, t2

# main
s, l, t0, t1, t2 = ff()

t4 = tm.time()    # time measurement

print(f"S:{s*1E-5:.0f}", end='')
print(f" | L:{l:0g}", end='')
print(f" | T1:{t1-t0:.4f}", end='')
print(f" | TF:{t2-t1:.4f}", end='')
print(f" | TT:{t2-t0:.4f}", end='')
print(f" | TO:{t4-t3:.4f}")
```

D.2.10 Parallel Numba-CPU

Listing D.29 - Parallel Numba-CPU implementation of the FFT test case.

```
import numpy as np, time as tm
from numba import njit, objmode
from mpi4py_fft import PFFT, newDistArray
from mpi4py import MPI

t3 = tm.time()    # time measurement

def uu() :
    return newDistArray(f, False)
def uf(u) :
    return f.forward(u, normalize=False)
@njit
def ff() :
    with objmode(t0 = 'f8') :
        t0 = tm.time()    # time measurement

    # data
    with objmode(u = 'complex128[:,:,:]') :  # annotate return type
        u = uu()
    for k in range (u.shape[2]) :
        for j in range(u.shape[1]) :
            for i in range(u.shape[0]) :
                u[i, j, k] = complex( np.sin ( i + j + k + 3 ), 0 )

    with objmode(t1 = 'f8') :
        t1 = tm.time()    # time measurement

    # FFT
    with objmode(u_hat = 'complex128[:,:,:]') :  # annotate return type
        u_hat = uf(u)
    # checksum
    s = np.array(np.sum(u_hat), dtype=np.complex128)
    rs = np.array(0, dtype=np.complex128)
    with objmode() :
        MPI.COMM_WORLD.Reduce([s, MPI.DOUBLE_COMPLEX],
            [rs, MPI.DOUBLE_COMPLEX], op=MPI.SUM, root=0)

    with objmode(t2 = 'f8') :
        t2 = tm.time()    # time measurement

    return rs, t0, t1, t2

# main
```

```python
ms = MPI.COMM_WORLD.Get_size()
mr = MPI.COMM_WORLD.Get_rank()
L = M = N = 576
# PFFT should be outside the numba function due to the "class" return
f = PFFT(MPI.COMM_WORLD, [N, M, L], dtype=np.complex128,
         backend='pyfftw')
# numba function
s, t0, t1, t2 = ff()

t4 = tm.time()    # time measurement

if not mr :
    print(f"S:{s*1E-5:.0f}", end='')
    print(f", L:{L:0g}", end='')
    print(f", N:{ms:0g}", end='')
    print(f", T1:{t1-t0:.4f}", end='')
    print(f", TF:{t2-t1:.4f}", end='')
    print(f", TT:{t2-t0:.4f}", end='')
    print(f", TO:{t4-t3:.4f}")
```

D.2.11 CuPY

Listing D.30 - CuPY implementation of the FFT test case.

```python
import numpy as np, cupy as cp, time as tm
def f() :
    t0 = -tm.time()    # <--- time measurement
    L = M = N = 576
    a = np.fromfunction( lambda i, j, k:
            np.sin ( i + j + k + 3 ), (N, M, L), dtype=cp.complex128 )
    f = cp.asarray(a)
    fft = cp.fft.fftn(f)
    s = complex(cp.sum(fft))
    t0 += tm.time()    # <--- time measurement
    print(f"S:{s*1e-5:.0f}", end='')
    print(f" | T:{t0:.4f}")
```

D.3 Implementations of the random forest test case

D.3.1 Serial and parallel F90

The code of the F90 implementation of the RF case study reuses the code from the PARF library, and below are just the modifications made to the library, both in the serial and parallel versions, which basically consists of measuring the processing time using the wall time, and in the case of the parallel version also show the number of MPI processes used.

Listing D.31 - Serial F90 implementation of the RF test case.

```fortran
PROGRAM random_forest

... original code at the beginning of PARF F90 code ...

  !=[ added code ]------------------------
  real :: t0, t1
  call cpu_time(t0)  ! time measurement
  !---------------------------------------

... original main PARF F90 code ...

  !=[ added code ]------------------------
  call cpu_time(t1)  ! time measurement
  if (par_rank == 0) then
    write(6, "('T: 'sf0.4'  |  N: 'g0)" ) t1-t0, par_processes
  endif
  !---------------------------------------

END PROGRAM random_forest
```

D.3.2 Serial and parallel F2PY

As with the F90 implementation, the F90 code from PARF is reused. Below are just the changed parts. Existing F90 code is repurposed and inserted into a subroutine as per the F2PY API specification. Basically library code changes are to make outputs or input values to be placed in main subroutine call parameters. It also implies that other internal subroutines are changed to include passing parameters that must reach the main subroutine.

Listing D.32 - Serial F2PY implementation of the RF test case - F90 module code.

```fortran
! the existing F90 code is repurposed and inserted into a subroutine.
!-[ changed ]----------------------------
! PROGRAM random_forest
SUBROUTINE random_forest(p_trainset, p_testset, &
                p_error_count, p_oob_count, p_kappa_value, &
                p_instance_count, p_error, p_testset_kappa_value, &
                p_time, p_rank, p_size)
!---------------------------------------

```

```
... inital PARF F90 code ...

!-[ changed ]----------------------------

! files
character(len=256), intent(in) :: p_trainset, p_testset

! Trainset
integer, intent(out) :: p_error_count, p_oob_count
real,    intent(out) :: p_kappa_value

! Testset
integer, intent(out) :: p_instance_count
real,    intent(out) :: p_error, p_testset_kappa_value

! Proc, time
integer, intent(out) :: p_rank, p_size
real,    intent(out) :: p_time

real :: t0, t1

p_error_count = 0
p_oob_count = 0
p_instance_count = 0
p_kappa_value = 0
p_error = 0
p_testset_kappa_value = 0

call cpu_time(t0)  ! time measurement
!----------------------------------------

... PARF F90 code ...

! Basically library code changes are to make outputs or input values to
! be placed in main subroutine call parameters. It also implies that
! other internal subroutines are changed to include passing parameters
! that must reach the main subroutine.
!-[ changed ]----------------------------
        CALL classify_instanceset(testset, rfptr, p_error, &
                    p_instance_count, p_testset_kappa_value)
!---------------------------------------

... PARF F90 code ...

!-[ changed ]----------------------------
```

```
            CALL calc_training_error(trainset, p_error_count, &
                         p_oob_count, p_kappa_value)
!----------------------------------------

... PARF F90 code ...

!-[ changed ]----------------------------
              CALL classify_instanceset(testset, rfptr, p_error, &
                         p_instance_count, p_testset_kappa_value)
!---------------------------------------

... PARF F90 code ...

!-[ changed ]----------------------------
        CALL classify_instanceset(protoset, rfptr, p_error, &
                     p_instance_count, p_testset_kappa_value)
!---------------------------------------

... PARF F90 code ...

!-[ changed ]----------------------------
call cpu_time(t1)  ! time measurement
p_size = par_processes
p_rank = par_rank
p_time = t1 - t0
!---------------------------------------

... PARF F90 code ...

! the existing F90 code is repurposed and inserted into a subroutine.
!-[ changed ]----------------------------
! END PROGRAM random_forest
END SUBROUTINE
!---------------------------------------
```

Listing D.33 - Serial F2PY implementation of the RF test case - Python main code.

```
import time as tm, parf003ser

t0 = tm.time()    # time measurement

resu = parf003ser.random_forest(
    "datasets/asteroid-train-66k.arff",
    "datasets/asteroid-test-34k.arff"
)
```

```python
p_error_count = resu[0]
p_oob_count = resu[1]
p_kappa_value = resu[2]
p_instance_count = resu[3]
p_error = resu[4]
p_testset_kappa_value = resu[5]
p_time = resu[6]
p_rank = resu[7]
p_size = resu[8]

t1 = tm.time()    # time measurement

if p_rank == 0 :
    print(f'Trainset classification error is',
          f'{p_error_count * 100 / p_oob_count :.2f}%',
          f'of {p_oob_count} (kappa: {p_kappa_value :.4f})')
    print(f' Testset classification error is {p_error * 100 :.2f}%',
          f'of {p_instance_count} (kappa: {p_testset_kappa_value :.4f})')
    print(f'T: {p_time :.4f}  |  N: {p_size :0g}')
```

D.3.3 Parallel F2PY

The parallel F90 version reuses the code from the PARF library which is built in two different ways, one for the serial version and one for the parallel, changing settings in the library before the build, and F2PY builds two Python libraries, one for the serial version and another for the parallel version. This Python library is then used in the main Python code of the F2PY implementation.

Listing D.34 - Parallel F2PY implementation of the RF test case - Python main code.

```python
import time as tm, parf003mpi

t0 = tm.time()    # time measurement

resu = parf003mpi.random_forest(
    "datasets/asteroid-train-66k.arff",
    "datasets/asteroid-test-34k.arff"
)
p_error_count = resu[0]
p_oob_count = resu[1]
p_kappa_value = resu[2]
p_instance_count = resu[3]
p_error = resu[4]
p_testset_kappa_value = resu[5]
```

```
p_time = resu[6]
p_rank = resu[7]
p_size = resu[8]

t1 = tm.time()    # time measurement

if p_rank == 0 :
    print(f'Trainset classification error is',
          f'{p_error_count * 100 / p_oob_count :.2f}%',
          f'of {p_oob_count} (kappa: {p_kappa_value :.4f})')
    print(f' Testset classification error is {p_error * 100 :.2f}%',
          f'of {p_instance_count} (kappa: {p_testset_kappa_value :.4f})')
    print(f'T: {p_time :.4f}  |  N: {p_size :0g}')
```

D.3.4 Serial Python

Listing D.35 - Serial Python implementation of the RF test case.

```
import pandas as pd
import numpy as np
import sys
from scipy.io import arff
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
from time import time
t = time()

data = arff.loadarff(sys.argv[1])
df = pd.DataFrame(data[0])
df = df.replace(b'N', 0)
df = df.replace(b'Y', 1)
df['class'] = df['class'].str.decode('utf-8').fillna(df['class'])
y_train = df['class']
X_train = df.drop(columns=['class'])
imp = SimpleImputer(missing_values = np.nan, strategy = 'mean')
df2 = pd.DataFrame(imp.fit_transform(X_train))
df2.columns = X_train.columns
df2.index = X_train.index
X_train = df2

datat = arff.loadarff(sys.argv[2])
df = pd.DataFrame(datat[0])
df = df.replace(b'N', 0)
df = df.replace(b'Y', 1)
```

```
df['class'] = df['class'].str.decode('utf-8').fillna(df['class'])
y_test = df['class']
X_test = df.drop(columns = ['class'])
imp = SimpleImputer(missing_values = np.nan, strategy = 'mean')
df2 = pd.DataFrame(imp.fit_transform(X_test))
df2.columns = X_test.columns
df2.index = X_test.index
X_test = df2

clf = RandomForestClassifier(n_estimators = 100)
clf.fit(X_train, y_train)
y_pred_test  = clf.predict(X_test)
y_pred_train = clf.predict(X_train)
accu = metrics.accuracy_score(y_train, y_pred_train, normalize = False)
trsi = y_train.size
perr = ((trsi - accu) / (trsi)) * 100
kapp = metrics.cohen_kappa_score(y_train, y_pred_train)
print(f'Trainset classification error is {perr:.2f}% ',
      f'of {trsi} (kappa: {kapp:.4f})')
accu = metrics.accuracy_score(y_test, y_pred_test, normalize = False)
trsi = y_test.size
perr = ((trsi - accu) / (trsi)) * 100
kapp = metrics.cohen_kappa_score(y_test, y_pred_test)
print(f' Testset classification error is {perr:.2f}% ',
      f'of {trsi} (kappa: {kapp:.4f})')

t = time() - t
print(f"T: {t:.4f} s")
```

D.3.5 Parallel Python

Listing D.36 - Parallel Python implementation of the RF test case.

```
import argparse, logging, os, sys, datetime, pandas as pd, numpy as np
from joblib import Parallel, parallel_backend, register_parallel_backend
from joblib import delayed, cpu_count
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
from scipy.io import arff
import ipyparallel as ipp
from ipyparallel.joblib import IPythonParallelBackend
from time import time
t = time()

```

```
# Prepare the engines
c = ipp.Client(profile = sys.argv[3])
ncli = len(c.ids)
bview = c.load_balanced_view()
register_parallel_backend(
    'ipyparallel',
    lambda : IPythonParallelBackend(view = bview))

# Get & prepare data
data = arff.loadarff(sys.argv[1])
df = pd.DataFrame(data[0])
df = df.replace(b'N', 0)
df = df.replace(b'Y', 1)
df['class'] = df['class'].str.decode('utf-8').fillna(df['class'])
y_train = df['class']
X_train = df.drop(columns=['class'])
imp = SimpleImputer(missing_values = np.nan, strategy = 'mean')
df2 = pd.DataFrame(imp.fit_transform(X_train))
df2.columns = X_train.columns
df2.index = X_train.index
X_train = df2

datat = arff.loadarff(sys.argv[2])
df = pd.DataFrame(datat[0])
df = df.replace(b'N', 0)
df = df.replace(b'Y', 1)
df['class'] = df['class'].str.decode('utf-8').fillna(df['class'])
y_test = df['class']
X_test = df.drop(columns = ['class'])
imp = SimpleImputer(missing_values = np.nan, strategy = 'mean')
df2 = pd.DataFrame(imp.fit_transform(X_test))
df2.columns = X_test.columns
df2.index = X_test.index
X_test = df2

clf = RandomForestClassifier(n_estimators = 100)
with parallel_backend('ipyparallel') :
    clf.fit(X_train, y_train)

y_pred_test  = clf.predict(X_test)
y_pred_train = clf.predict(X_train)
accu = metrics.accuracy_score(y_train, y_pred_train, normalize = False)
trsi = y_train.size
perr = ((trsi - accu) / (trsi)) * 100
kapp = metrics.cohen_kappa_score(y_train, y_pred_train)
```

```python
print(f'Trainset classification error is {perr:.2f}% ',
      f'of {trsi} (kappa: {kapp:.4f})')
accu = metrics.accuracy_score(y_test, y_pred_test, normalize = False)
trsi = y_test.size
perr = ((trsi - accu) / (trsi)) * 100
kapp = metrics.cohen_kappa_score(y_test, y_pred_test)
print(f' Testset classification error is {perr:.2f}% ',
      f'of {trsi} (kappa: {kapp:.4f})')

t = time() - t
print(f"T: {t:.4f}  |  N: {ncli:0g}")

c.shutdown(hub=True, block=False)
```

D.3.6 Serial Cython

Listing D.37 - Serial Cython implementation of the RF test case - Cython module code.

```python
#cython: boundscheck=False, wraparound=False, cdivision=True
#cython: initializedcheck=False, language_level=3, infer_types=True
def rfcsf(trainset, testset) :
    import pandas as pd
    import numpy as np
    import sys
    from scipy.io import arff
    from sklearn.impute import SimpleImputer
    from sklearn.ensemble import RandomForestClassifier
    from sklearn import metrics

    data = arff.loadarff(trainset)
    df = pd.DataFrame(data[0])
    df = df.replace(b'N', 0)
    df = df.replace(b'Y', 1)
    df['class'] = df['class'].str.decode('utf-8').fillna(df['class'])
    y_train = df['class']
    X_train = df.drop(columns=['class'])
    imp = SimpleImputer(missing_values = np.nan, strategy = 'mean')
    df2 = pd.DataFrame(imp.fit_transform(X_train))
    df2.columns = X_train.columns
    df2.index = X_train.index
    X_train = df2

    datat = arff.loadarff(testset)
    df = pd.DataFrame(datat[0])
    df = df.replace(b'N', 0)
    df = df.replace(b'Y', 1)
    df['class'] = df['class'].str.decode('utf-8').fillna(df['class'])
    y_test = df['class']
    X_test = df.drop(columns = ['class'])
    imp = SimpleImputer(missing_values = np.nan, strategy = 'mean')
    df2 = pd.DataFrame(imp.fit_transform(X_test))
    df2.columns = X_test.columns
    df2.index = X_test.index
    X_test = df2

    clf = RandomForestClassifier(n_estimators = 100)
    clf.fit(X_train, y_train)
    y_pred_test  = clf.predict(X_test)
    y_pred_train = clf.predict(X_train)
    accu = metrics.accuracy_score(y_train, y_pred_train, normalize = False)
    trtrsi = y_train.size
    trperr = ((trtrsi - accu) / (trtrsi)) * 100
    trkapp = metrics.cohen_kappa_score(y_train, y_pred_train)

    accu = metrics.accuracy_score(y_test, y_pred_test, normalize = False)
    tetrsi = y_test.size
    teperr = ((tetrsi - accu) / (tetrsi)) * 100
    tekapp = metrics.cohen_kappa_score(y_test, y_pred_test)

    return trtrsi, trperr, trkapp, tetrsi, teperr, tekapp
```

Listing D.38 - Serial Cython implementation of the RF test case - Python main code.

```
from time import time
from rfcs import rfcsf

t0 = time()
trainset = "datasets/asteroid-train-66k.arff"
testset  = "datasets/asteroid-test-34k.arff"
trtrsi, trperr, trkapp, tetrsi, teperr, tekapp = rfcsf(trainset, testset)
t1 = time() - t0
print(f'Trainset classification error is {trperr:.2f}% ',
      f'of {trtrsi} (kappa: {trkapp:.4f})')
print(f' Testset classification error is {teperr:.2f}% ',
      f'of {tetrsi} (kappa: {tekapp:.4f})')
print(f"T: {t1:.4f}")
```

D.3.7 Parallel Cython

Listing D.39 - Parallel Cython implementation of the RF test case - Cython module.

```
#cython: boundscheck=False, wraparound=False, cdivision=True
#cython: initializedcheck=False, language_level=3, infer_types=True
def rfcmf(trainset, testset) :
    import logging, os, sys, datetime
    import pandas as pd, numpy as np
    from sklearn.impute import SimpleImputer
    from sklearn.ensemble import RandomForestClassifier
    from sklearn import metrics
    from scipy.io import arff
    import ipyparallel as ipp
    from ipyparallel.joblib import IPythonParallelBackend
    from joblib import Parallel, parallel_backend
    from joblib import register_parallel_backend
    from joblib import delayed, cpu_count
    from time import time
    t = time()

    # Get & prepare data
    data = arff.loadarff(trainset)
    df = pd.DataFrame(data[0])
    df = df.replace(b'N', 0)
    df = df.replace(b'Y', 1)
    df['class'] = df['class'].str.decode('utf-8').fillna(df['class'])
    y_train = df['class']
    X_train = df.drop(columns=['class'])
    imp = SimpleImputer(missing_values = np.nan, strategy = 'mean')
    df2 = pd.DataFrame(imp.fit_transform(X_train))
    df2.columns = X_train.columns
    df2.index = X_train.index
    X_train = df2

    datat = arff.loadarff(testset)
    df = pd.DataFrame(datat[0])
    df = df.replace(b'N', 0)
    df = df.replace(b'Y', 1)
    df['class'] = df['class'].str.decode('utf-8').fillna(df['class'])
    y_test = df['class']
    X_test = df.drop(columns = ['class'])
    imp = SimpleImputer(missing_values = np.nan, strategy = 'mean')
    df2 = pd.DataFrame(imp.fit_transform(X_test))
    df2.columns = X_test.columns
    df2.index = X_test.index
    X_test = df2

    clf = RandomForestClassifier(n_estimators = 100)
    with parallel_backend('ipyparallel') :
        clf.fit(X_train, y_train)
    y_pred_test  = clf.predict(X_test)
    y_pred_train = clf.predict(X_train)
    accu = metrics.accuracy_score(y_train, y_pred_train,
                                  normalize = False)
    trtrsi = y_train.size
    trperr = ((trtrsi - accu) / (trtrsi)) * 100
    trkapp = metrics.cohen_kappa_score(y_train, y_pred_train)

    accu = metrics.accuracy_score(y_test, y_pred_test,
                                  normalize = False)
    tetrsi = y_test.size
    teperr = ((tetrsi - accu) / (tetrsi)) * 100
    tekapp = metrics.cohen_kappa_score(y_test, y_pred_test)

    return trtrsi, trperr, trkapp, tetrsi, teperr, tekapp
```

Listing D.40 - Parallel Cython implementation of the RF test case - Python main code.

```
import argparse
from time import time
from rfcm import rfcmf
import ipyparallel as ipp
from ipyparallel.joblib import IPythonParallelBackend
from joblib import Parallel, parallel_backend
from joblib import register_parallel_backend
from joblib import delayed, cpu_count

t0 = time()
trainset = "datasets/asteroid-train-66k.arff"
testset  = "datasets/asteroid-test-34k.arff"
parser = argparse.ArgumentParser()
parser.add_argument("-p", "--profile", required=True,
    help="Name of IPython profile to use")
profile = parser.parse_args().profile

# Prepare the engines
c = ipp.Client(profile = profile)
ncli = len(c.ids)
bview = c.load_balanced_view()
register_parallel_backend('ipyparallel',
    lambda : IPythonParallelBackend(view = bview) )
```

```python

( trtrsi, trperr, trkapp, tetrsi, teperr, tekapp
    ) = rfcmf(trainset, testset)

# Shutdown the engines
c.shutdown(hub=True, block=False)

# Result
t1 = time() - t0
print(f'Trainset classification error is {trperr:.2f}% ',
      f'of {trtrsi} (kappa: {trkapp:.4f})')
print(f' Testset classification error is {teperr:.2f}% ',
      f'of {tetrsi} (kappa: {tekapp:.4f})')
print(f"T: {t1:.4f}  |  N: {ncli:0g}")
```

D.3.8 Serial Numba-CPU

Listing D.41 - Serial Numba-CPU implementation of the RF test case.

```python
import pandas as pd
import numpy as np
import sys
from scipy.io import arff
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
from numba import jit, objmode
from time import time
t0 = time()

@jit(forceobj=True)
def rfcsf(trainset, testset) :
    data = arff.loadarff(trainset)
    df = pd.DataFrame(data[0])
    df = df.replace(b'N', 0)
    df = df.replace(b'Y', 1)
    df['class'] = df['class'].str.decode('utf-8').fillna(df['class'])
    y_train = df['class']
    X_train = df.drop(columns=['class'])
    imp = SimpleImputer(missing_values = np.nan, strategy = 'mean')
    df2 = pd.DataFrame(imp.fit_transform(X_train))
    df2.columns = X_train.columns
    df2.index = X_train.index
    X_train = df2

    datat = arff.loadarff(testset)
    df = pd.DataFrame(datat[0])
    df = df.replace(b'N', 0)
    df = df.replace(b'Y', 1)
    df['class'] = df['class'].str.decode('utf-8').fillna(df['class'])
    y_test = df['class']
    X_test = df.drop(columns = ['class'])
    imp = SimpleImputer(missing_values = np.nan, strategy = 'mean')
    df2 = pd.DataFrame(imp.fit_transform(X_test))
    df2.columns = X_test.columns
    df2.index = X_test.index
    X_test = df2

    clf = RandomForestClassifier(n_estimators = 100)
    clf.fit(X_train, y_train)
    y_pred_test  = clf.predict(X_test)
    y_pred_train = clf.predict(X_train)
    accu = metrics.accuracy_score(y_train, y_pred_train, normalize = False)
    trtrsi = y_train.size
    trperr = ((trtrsi - accu) / (trtrsi)) * 100
    trkapp = metrics.cohen_kappa_score(y_train, y_pred_train)

    accu = metrics.accuracy_score(y_test, y_pred_test, normalize = False)
    tetrsi = y_test.size
    teperr = ((tetrsi - accu) / (tetrsi)) * 100
    tekapp = metrics.cohen_kappa_score(y_test, y_pred_test)

    return trtrsi, trperr, trkapp, tetrsi, teperr, tekapp

# main
trainset = "datasets/asteroid-train-66k.arff"
testset  = "datasets/asteroid-test-34k.arff"
trtrsi, trperr, trkapp, tetrsi, teperr, tekapp = rfcsf(trainset, testset)
t1 = time() - t0
print(f'Trainset classification error is {trperr:.2f}% ',
      f'of {trtrsi} (kappa: {trkapp:.4f})')
print(f' Testset classification error is {teperr:.2f}% ',
      f'of {tetrsi} (kappa: {tekapp:.4f})')
print(f"T: {t1:.4f}")
```

D.3.9 Parallel Numba-CPU

Listing D.42 - Parallel Numba-CPU implementation of the RF test case.

```
import argparse, logging, os, sys, datetime
```

```python
import pandas as pd, numpy as np
from joblib import ( Parallel, parallel_backend,
                     register_parallel_backend )
from joblib import delayed, cpu_count
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
from scipy.io import arff
import ipyparallel as ipp
from ipyparallel.joblib import IPythonParallelBackend
from numba import jit, objmode
from time import time
t0 = time()

def eng01(clf, X_train, y_train) :
    with parallel_backend('ipyparallel') :
        clf.fit(X_train, y_train)
    return clf

@jit(forceobj=True)
def rfamf(trainset, testset) :

    # Get & prepare data
    data = arff.loadarff(trainset)
    df = pd.DataFrame(data[0])
    df = df.replace(b'N', 0)
    df = df.replace(b'Y', 1)
    df['class'] = df['class'].str.decode('utf-8').fillna(df['class'])
    y_train = df['class']
    X_train = df.drop(columns=['class'])
    imp = SimpleImputer(missing_values = np.nan, strategy = 'mean')
    df2 = pd.DataFrame(imp.fit_transform(X_train))
    df2.columns = X_train.columns
    df2.index = X_train.index
    X_train = df2

    datat = arff.loadarff(testset)
    df = pd.DataFrame(datat[0])
    df = df.replace(b'N', 0)
    df = df.replace(b'Y', 1)
    df['class'] = df['class'].str.decode('utf-8').fillna(df['class'])
    y_test = df['class']
    X_test = df.drop(columns = ['class'])
    imp = SimpleImputer(missing_values = np.nan, strategy = 'mean')
    df2 = pd.DataFrame(imp.fit_transform(X_test))
    df2.columns = X_test.columns
    df2.index = X_test.index
    X_test = df2

    clf = RandomForestClassifier(n_estimators = 100)
    clf = eng01(clf, X_train, y_train)
    y_pred_test  = clf.predict(X_test)
    y_pred_train = clf.predict(X_train)
    accu = metrics.accuracy_score(y_train, y_pred_train,
                                  normalize = False)
    trtrsi = y_train.size
    trperr = ((trtrsi - accu) / (trtrsi)) * 100
    trkapp = metrics.cohen_kappa_score(y_train, y_pred_train)

    accu = metrics.accuracy_score(y_test, y_pred_test,
                                  normalize = False)
    tetrsi = y_test.size
    teperr = ((tetrsi - accu) / (tetrsi)) * 100
    tekapp = metrics.cohen_kappa_score(y_test, y_pred_test)

    return trtrsi, trperr, trkapp, tetrsi, teperr, tekapp

# Main
trainset = "datasets/asteroid-train-66k.arff"
testset  = "datasets/asteroid-test-34k.arff"
parser = argparse.ArgumentParser()
parser.add_argument("-p", "--profile", required=True,
    help="Name of IPython profile to use")
profile = parser.parse_args().profile

# Prepare the engines
c = ipp.Client(profile = profile)
ncli = len(c.ids)
bview = c.load_balanced_view()
register_parallel_backend('ipyparallel',
    lambda : IPythonParallelBackend(view = bview) )

# Call Numba Code
( trtrsi, trperr, trkapp, tetrsi, teperr, tekapp,
     ) = rfamf(trainset, testset)

# Shutdown the engines
c.shutdown(hub=True, block=False)

# Result
```

```
t1 = time() - t0
print(f'Trainset classification error is {trperr:.2f}% ',
      f'of {trtrsi} (kappa: {trkapp:.4f})')
print(f' Testset classification error is {teperr:.2f}% ',
      f'of {tetrsi} (kappa: {tekapp:.4f})')
print(f"T: {t1:.4f}  |  N: {ncli:0g}")
```

ANNEX A - STENCIL CODE

This annex 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 Serial version

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 3.1.1).

Listing A.1 - stencil.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 Parallel version

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>

```

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

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_ */
```