NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

MIGUEL MORATO ROXO RIBEIRO PINTO

BSc in Computer Science and Computer Engineering

PYTHON FRAMEWORK FOR PARALLELISM ON A CLUSTER

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

April, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

PYTHON FRAMEWORK FOR PARALLELISM ON A CLUSTER

MIGUEL MORATO ROXO RIBEIRO PINTO

BSc in Computer Science and Computer Engineering

Adviser: Doutor Vítor Manuel Alves Duarte
Assistant Professor, NOVA University Lisbon

Examination Committee

Chair: Doutora Cláudia Alexandra Magalhães Soares
Assistant Professor, NOVA University Lisbon

Rapporteur: Doutor Carlos Jorge de Sousa Gonçalves
Adjunct Professor, ISEL - Instituto Superior de Engenharia de Lisboa

Adviser: Doutor Vítor Manuel Alves Duarte
Assistant Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
April, 2024

Python framework for parallelism on a cluster

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.17) [1].

Acknowledgements

I am grateful to my parents, Ana Isabel Pinto and António Augusto Pinto, for their unwavering support. I would also like to express my gratitude towards my brother Tomás Pinto, my grandmother Lisete Roxo, my family, and friends who have provided me with strong moral support and accompanied me throughout this research. Their constructive criticism has been invaluable in improving my work.

Above all, I extend my heartfelt thanks to my advisor, Assistant Professor Vítor Duarte, for his constant availability, guidance, and for pointing me in the right direction to develop my dissertation. I am grateful to everyone mentioned above for their contributions. Thank you all very much.

Abstract

The increasing demand for high-performance computing and the need to process large amounts of data have made Distributed Parallel Programming an essential part of modern computing. The traditional sequential processing approach can no longer keep up with data's increasing volume and complexity. This is where parallel programming comes into play. By dividing a computational task into smaller, parallelizable tasks, we can use multiple processing units to achieve faster and more efficient processing. However, not all programming languages are well-suited for parallel programming.

Python has become a popular choice for many scientists and engineers due to its simplicity, readability, and versatility. Python is well-equipped to handle the demands of parallel processing in a cluster environment and several Python libraries, namely Torc_py, Scoop, and Dask, have been developed to further help the common programmer to explore the parallelism in multiprocessors and clusters. Those libraries offer simplicity and ease of use for common programming patters and also follow Python's straightforward syntax and user-friendly, making this language a great option for advanced programmers and also to developers who are new to parallel programming.

This research aims to explore the programming language Python and some of its libraries, provide a comprehensive understanding of the current state of parallel distributed programming in Python, and implement support for parallel patterns lacking in the current modules. In particular, we aim to contribute to the easier development of programs based on Torc_py that use patterns like stencil, convolution, and search that can abort and calculate functions on data.

Keywords: Distributed Parallel Programming, Python, Torc_py, Scoop, Dask

Resumo

A crescente procura por computação de alto desempenho assim como a necessidade de processar grandes quantidades de dados tornaram a Programação Paralela Distribuída uma parte essencial da computação moderna. A abordagem tradicional de processamento sequencial já não consegue acompanhar tanto o aumento do volume como a sua maior complexidade. É aqui que entra a Programação Paralela Distribuída. Ao dividir uma tarefa em pequenas tarefas paralelizáveis, podemos aproveitar as múltiplas unidades de processamento para alcançar um processamento mais rápido e eficiente.

No entanto, nem todas as linguagens de programação são adequadas à programação paralela. O Python, por exemplo, tornou-se uma escolha popular para muitos cientistas e engenheiros devido à sua simplicidade, legibilidade e versatilidade. O Python está bem desenhado para responder as exigências de processamento paralelo em ambiente de cluster. Python conta com várias bibliotecas disponíveis para paralelismo em cluster como, por exemplo, Torc_py, Scoop e Dask. Uma das razões pelas quais o Python é uma escolha popular para paralelismo em cluster é a sua simplicidade e facilidade de utilização. As características do Python, nomeadamente a sua simples sintaxe e bibliotecas amigas do utilizador, constituem uma ótima opção para programadores novos em programação paralela.

Esta pesquisa pretende explorar a linguagem de programação Python e algumas de suas bibliotecas, fornecer uma compreensão abrangente do estado atual da Programação Paralela Distribuída em Python e implementar suporte para padrões de programação paralela em falta nos módulos atuais. Em particular, pretende-se contribuir para um desenvolvimento simples de programas que utilizam padrões: stencil, convolução e de pesquisa que podem abortar e calculam funções sobre dados.

Palavras-chave: Programação Paralela Distribuída, Python, Torc_py, Scoop, Dask

Contents

List of Figures viii

List of Tables ix

1 Introduction 1
- 1.1 Contextualization . . . 1
  - 1.1.1 Python vs Matlab . . . 2
- 1.2 Problem . . . 3
- 1.3 Objectives . . . 3
- 1.4 Organization of the document . . . 4

2 Document Review 6
- 2.1 Programming patterns . . . 6
  - 2.1.1 Fork-Join . . . 6
  - 2.1.2 Map . . . 6
  - 2.1.3 Stencil . . . 7
  - 2.1.4 Reduction . . . 8
  - 2.1.5 Scan . . . 9
  - 2.1.6 Search . . . 10
  - 2.1.7 Gather . . . 10
  - 2.1.8 Scatter . . . 10
- 2.2 Python . . . 11
- 2.3 Python's libraries . . . 11
  - 2.3.1 Python's multiprocessing . . . 12
  - 2.3.2 Python's futures . . . 12
  - 2.3.3 Torc_Py . . . 13
  - 2.3.4 SCOOP . . . 15

2.3.5 Dask . . . 16
2.4 Proposed approach . . . 17

3 Implementation **19**
3.1 API Documentation . . . 19
3.2 Search Pattern . . . 22
3.3 Stencil Filter Pattern . . . 23
3.4 Integration and Workflow . . . 24
3.5 Conclusion . . . 24

4 Evaluation **26**
4.1 Experimental Setup . . . 26
4.1.1 Hardware Configuration . . . 26
4.1.2 Software Configuration . . . 27
4.2 Performance Analysis . . . 27
4.2.1 Experimental Metrics . . . 27
4.2.2 Performance and Scalability Analysis . . . 28
4.2.3 Comparison with Existing Patterns . . . 28
4.3 Testing Scenarios . . . 29
4.3.1 Search Pattern . . . 29
4.3.2 Stencil Pattern . . . 36

5 Conclusions **39**

Bibliography **41**

Annexes

I Annex 1 Search First Element Tables **44**

II Annex 2 Search Last Element Tables **46**

III Annex 3 Search Middle Element Tables **48**

IV Annex 4 Stencil Filter Tables **50**

List of Figures

2.1 The map pattern [8]. 7
2.2 The stencil pattern [9]. 7
2.3 The reduction pattern [11]. 9
2.4 The parallel reduction pattern [12]. 9
2.5 The scan pattern. (left is serial version while right is parallel version) [13]. 10

4.1 Search First Index Metrics 30
4.2 Search Last Index Metrics 32
4.3 Search Middle Index Metrics 34
4.4 Stencil Metrics 37

List of Tables

I.1 Time taken (in seconds) for each program to finish 44
I.2 Calculated metrics for mpi4py 44
I.3 Calculated metrics for torcpy's map 45
I.4 Calculated metrics for extended torcpy 45

II.1 Time taken (in seconds) for each program to finish 46
II.2 Calculated metrics for mpi4py 46
II.3 Calculated metrics for torcpy's map 47
II.4 Calculated metrics for extended torcpy 47

III.1 Time taken (in seconds) for each program to finish 48
III.2 Calculated metrics for mpi4py 48
III.3 Calculated metrics for torcpy's map 49
III.4 Calculated metrics for extended torcpy 49

IV.1 Time taken (in seconds) for each program to finish 50
IV.2 Calculated metrics for mpi4py 50
IV.3 Calculated metrics for torcpy's map 51
IV.4 Calculated metrics for extended torcpy 51

1 Introduction

1.1 Contextualization

The increasing demand for high-performance computing and the need to process large amounts of data have made distributed parallel programming an essential part of modern computing.

In the modern world of big data and complex algorithms, the need for efficient and scalable processing has become imperative. The traditional sequential processing approach can no longer keep up with data's increasing volume and complexity. This is where parallel programming comes into play. By dividing a computational task into smaller, parallelisable tasks, we can make use of multiple processing units to achieve faster and more efficient processing.

Parallel programming is even more critical in a cluster environment, as it allows us to harness the power of multiple machines working together. By leveraging the resources of various nodes in a cluster, we can easily tackle the most significant and most complex computational problems.

However, not all programming languages are well-suited for parallel programming. Python, on the other hand, has become a popular choice for many scientists and engineers due to its simplicity, readability, and versatility. It is well-equipped with several modules to handle the demands of parallel processing in a multi-core and cluster environment.

Several Python libraries are available for parallelism in a cluster, including `Torc_Py`, `Scoop`, and `Dask`. Python is a popular choice for parallelism in a cluster due to its simplicity and ease of use. It is known for its straightforward syntax and user-friendly libraries, making it an excellent option for developers new to parallel programming.

Other programming languages, like the popular Matlab, can also be considered to have strong support for numerical processing. Although Matlab provides

a parallel computing toolbox that enables you to run parallel and distributed computations in a Matlab session, either on local multi-core machines or on a cluster of computers, Matlab is limited in its ability to handle data types other than numbers. On the other hand, Python has built-in support for a wide range of data types, including strings, lists, and dictionaries, making it a more versatile option for parallel processing.

1.1.1 Python vs Matlab

Python and Matlab are both popular programming languages in the scientific and engineering community, but both have distinct features:

- Python is an open-source language, meaning it's free to use and has a large and active community of developers who contribute to its development and maintenance.
- Matlab is proprietary software developed by MathWorks and requires a license, although it offers a student version for a reduced fee.
- Python is one of the world's most widely used programming languages, and its popularity is increasing among data science and machine learning communities.
- Matlab's popularity has remained relatively stable over the years.
- Python has a vast ecosystem of libraries and tools that support various scientific and engineering tasks, including data analysis, visualisation, and machine learning. These libraries, such as NumPy, Pandas, and Matplotlib, are free of charge and maintained by a large and active community of developers.
- Matlab's toolboxes and libraries, although powerful, are proprietary and require a license to use.

Previous experiments at FCT/UNL with Directional Direct-Search optimization algorithms, written in Matlab and then re-implemented in Python[2] [3] have shown a viable path for future developments, without losing performance, for greater dissemination and usage of those algorithms. Additionally, Python has a large and active community of developers constantly working on improving and expanding its capabilities. In conclusion, while Python and Matlab have advantages, Python is a more versatile, widely used, and cost-effective option

for scientific and engineering applications, even when making use of parallel processing.

1.2 Problem

From the documentation review concerning Python and Matlab programming languages, it became apparent that common patterns should be available to improve and facilitate parallel programming. For example, in the case of image filtering or convolution, the stencil pattern is a requirement not provided by Torc_Py or Scoop as referred in chapter 2. In these patterns, the data distribution is not disjointed, and a portion of the data, corresponding to each point's neighbour (halo), must be distributed among several workers. This could be difficult for the programmer and a source of mistakes.

In the case of Directional Direct-Search Algorithms, the objective function to optimise is evaluated for various points in the domain, seeking to find better solutions for this function in a given area of the domain. With each best solution found, the process is refocused on this new point, always trying to find better solutions near this new point, guided by heuristics. This process is repeated iteratively, converging to the best solution or until reaching a specific stopping trait (margin of error or a limited number of iterations). An optimisation of these algorithms can terminate each iteration and move on to the next one as soon as a better solution is found, avoiding continuing the estimates for the remaining points. Examples of these algorithms are the SID-PSM (Simplex Derivatives in Pattern Search Methods) and DMS (Direct Multi Search) [4] [5] [6], developed at the Department of Mathematics of FCT/UNL.

1.3 Objectives

This research aims to assess and enhance Python's Futures API to accommodate various usage patterns. This includes enabling searches in Directional Direct-Search and Stencil/Convolution, which are not always supported in popular libraries such as **Torc_Py**, **Scoop**, and **Dask**. The study will delve into these libraries and their relationships and investigate the distinctive features of these commonly used libraries for distributed parallel computing in Python.

Also, this research aims to explore these libraries, provide a comprehensive understanding of the current state of parallel distributed programming in Python, try to fix the identified problems and study the performance of the implemented

extensions with simple and not-so-simple problems. It is intended to improve support for new patterns (for example the search pattern with the ability to stop and inform other tasks that it has found a value) and facilitate the correct development of applications like those discussed above.

As will be discussed in the next chapter, Torc_py is a top contender for parallel programming. It is well-built and properly documented and can abstract the use of the MPI interface, allowing the deployment and exploration of big clusters. This library could accommodate some improvements to adequately support the discussed patterns, contributing to its user community.

So, the plan to follow includes these steps:

- Implement the common stencil pattern, which is lacking;
- Implement a searching pattern with the ability to stop when some condition is met. That condition can express having found some value or, more generally, having found some element that evaluates a provided function to True. When found in one worker, all the other works can stop, instead of wasting resources, therefore, also as an objective will be implementing a stopping condition for these kind of pattern.

As part of the objectives of this research, a convenient API that extends Torc_Py with the described patterns is also proposed. Finally, an implementation is described and evaluated to assess the developed solution.

1.4 Organization of the document

This document is organized into five main sections, with each one concentrating on a vital aspect of the research. It has been structured in such a manner that it leads the reader from the background information, methodology followed, to implementation then evaluation to conclusions.

- **Section 1: Introduction** - It gives an overview of our research topic, identifies the objectives and states what problem the research team investigated. It is the context within which the entire research is conducted, giving reasons for conducting the research.
- **Section 2: Document Review** - Provides a comprehensive review of the existing literature and relevant documents. It establishes the theoretical and empirical foundation for the research, identifying key concepts, methodologies, and gaps that this dissertation aims to address.

- **Section 3: Implementation** - The practical aspects of the research are detailed in this section. It covers the design and development of the proposed solution, including the tools, techniques, and procedures employed. This section also discusses the challenges faced during implementation and how they were overcome.
- **Section 4: Evaluation** - In this section, it is evaluated the effectiveness and efficiency of the implemented solution. It presents the criteria for evaluation, the methodology used for data collection and analysis, and the results obtained.
- **Section 5: Conclusions** - In the final section it is summarised the key findings of the research.

Each section builds upon the previous one, ensuring a logical flow and a coherent narrative that collectively presents a thorough investigation into the research topic.

2

Document Review

This chapter covers key programming patterns and Python libraries. It starts by looking at important programming patterns used in parallel and distributed computing such as Stencil, Map, Reduction and Search. Then, it is explored some Python libraries like multiprocessing, futures, and Torc_Py, which make parallel processing and concurrent execution easier.

2.1 Programming patterns

A pattern in computer science refers to a standard solution to a recurring problem. Patterns can be applied to many different areas of software development, including data structures, algorithms, and parallel programming. In parallel programming, patterns provide a way to organise and structure code to take advantage of parallel computing resources, such as multiple processors or cores. Using patterns, developers can leverage well-established solutions to common parallel programming problems, reducing the time and effort needed to design and implement new solutions.

2.1.1 Fork-Join

The fork–join pattern lets control flow fork into multiple parallel flows that rejoin later [7]. In parallel programming, it is used to split a problem into smaller chunks that may be worked on simultaneously, reducing the overall processing time.

2.1.2 Map

The map pattern is a technique in parallel programming that replicates a function over every element in an index set [7].

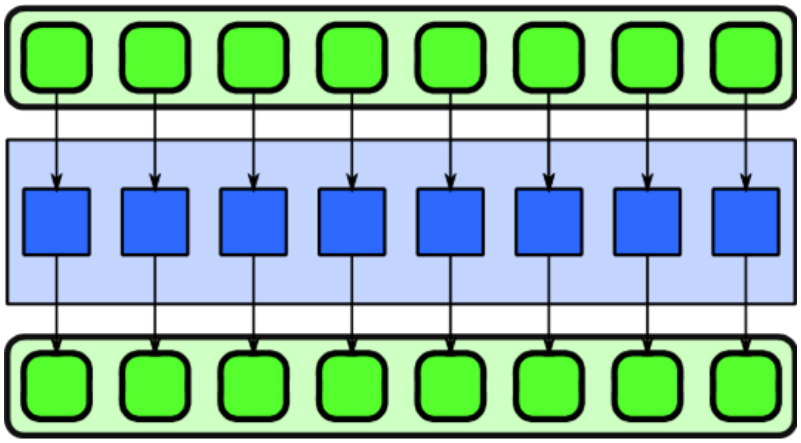


Figure 2.1: The map pattern [8].

The index set can be abstract or associated with the elements of a collection. The function being replicated is known as the elemental function, and it applies to the elements of an actual collection of input data. The map pattern replaces specific uses of iteration in serial programs, such as a loop in which every iteration is independent, the number of iterations is known in advance, and the computation depends only on the iteration count and data read from the collection. The elemental function must be pure, meaning it has no side effects, so the map pattern can be implemented in parallel and produce deterministic results. Examples of map pattern use include image processing, colour space conversions, Monte Carlo sampling, and ray tracing.

2.1.3 Stencil

The stencil pattern is an extension of the map pattern in which an elemental function can access not only one element in an input collection but also a set of neighbours defined by a set of relative offsets.

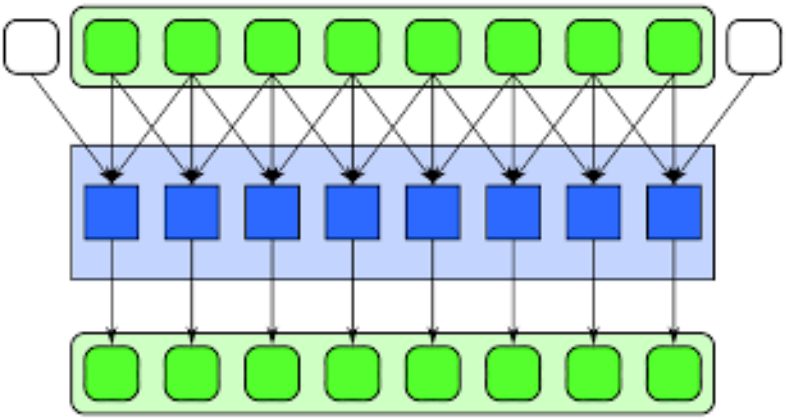


Figure 2.2: The stencil pattern [9].

The stencil pattern's efficient implementation, which uses tiling, makes data reuse possible. The stencil pattern is frequently employed in conjunction with iteration for picture filtering, simulation, and many other linear algebraic tasks. The implementation must avoid utilising special-case code inside the index domain and consider boundary conditions for array accesses. Applications for the stencil pattern include partial differential equation solvers, fluid flow simulation, and picture filtering.

2.1.3.1 Convolution matrix

Given that it is a frequently employed operation in image processing and computer vision, the convolution matrix computation can be used based on the stencil pattern. A convolution matrix itself is not necessarily a parallel pattern. However, in order to increase performance, the computation of a convolution matrix might be carried out in parallel. The convolution matrix is a tool for performing image processing operations by applying a matrix of values to an image, as stated in the GIMP documentation [10]. A set of coefficients determining the kind of operation carried out on the image is defined to produce the matrix. The convolution matrix is then applied to the image by sliding it over it, multiplying the relevant image elements by the elements of the convolution matrix element by element, and adding the results. After that, a new image of the same size as the original is produced, with each pixel representing the sum of the original image's elements times their corresponding values in the convolution matrix. This technique is frequently utilised for applications like picture sharpening, edge detection, image processing and computer vision activities.

2.1.4 Reduction

A reduction combines every element in a collection into a single element using an associative combiner function [7].

A reduction can be parallelised using a tree structure and can be used to find the sum of elements in a collection. It can also be used for operations like averaging, testing convergence, image comparison, dot products, and matrix multiplication.

The pattern has various applications in numerical applications, Monte Carlo samples, iterative solutions of systems of linear equations, video encoding, and matrix multiplication. However, care should be taken when using operations that are not genuinely associative, such as floating point addition, as different orderings can result in different results.

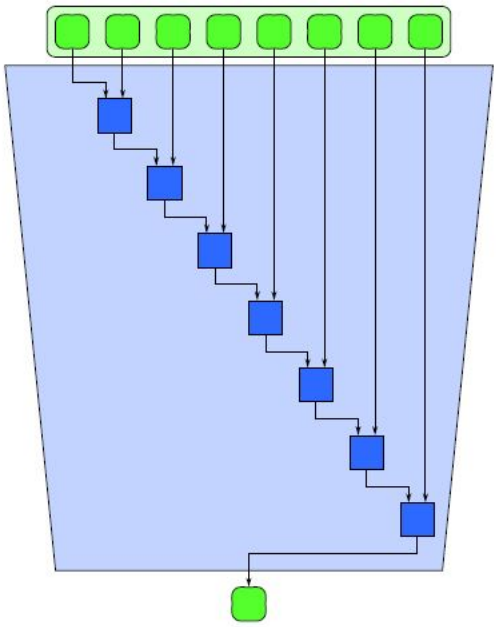


Figure 2.3: The reduction pattern [11].

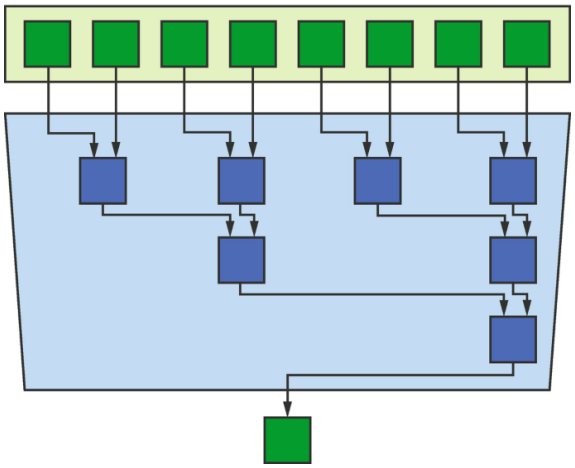


Figure 2.4: The parallel reduction pattern [12].

2.1.5 Scan

By computing a reduction of the input up to each output location, Scan computes all the intermediate reductions of a collection [7]. Scan is typically a specific case of a serial pattern known as a fold. It is possible to rearrange and parallelise an associative successor function used in a fold. The associative successor function in the text example is addition. Although less effective and maybe more labour-intensive than the serial approach, Scan can also be implemented in parallel. Integration, decision simulations, and random number creation are a few instances of applications for Scan.

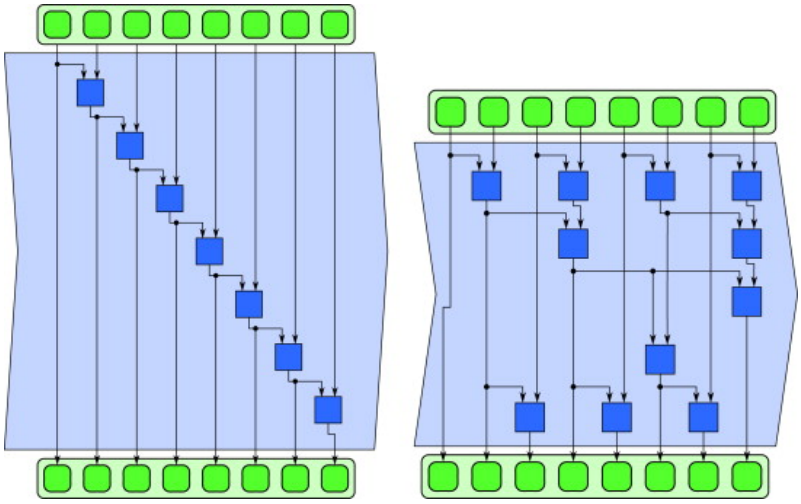


Figure 2.5: The scan pattern. (left is serial version while right is parallel version) [13].

2.1.6 Search

Given a collection, the search pattern finds data that meets some criteria [7]. The Search pattern refers to the process of finding elements within a collection that match specific criteria. The criteria can range from simple key-value associations to more complex combinations of logical and arithmetic constraints. The data may be maintained in a sorted order to improve search efficiency, but other implementation methods can also be used.

2.1.7 Gather

The Gather pattern is a way of reading data from one collection and putting it into another, based on a set of indices [7]. The element type of the output data collection is the same as the input data collection, and it combines a Map operation with random serial read operations. Gather can be made more efficient in some situations where the indices exhibit well-defined patterns, such as when changing data within an array. Collision detection, ray tracing, volume rendering, proximity searches, and sparse matrix operations are just a few of the uses for gather.

2.1.8 Scatter

The scatter pattern writes data to a collection at specified locations, which are given as a set of indices [7]. It is the reverse of the gather pattern. With scatter, it is difficult to determine which value should be written if two input data items are written to the same spot, which results in a collision. Rules for deterministically

resolving collisions must be established in order to avoid this problem. These rules might involve combining values with associative operators, picking a value randomly, or prioritising values.

2.2 Python

Python is a high-level general-purpose programming language [14]. Python is appropriate for use as a scripting language for implementing web applications. Python has the performance required for even compute-intensive workloads because it can be extended in C and C++. Thanks to its strong organisational features and consistent use of object-oriented programming, we can design straightforward, logical applications for tasks.

Python's main features are:

- Built-in high-level data types.
- Standard control structures, including "while", "if", "if-else", "if-elif-else", and a robust collection iterator "for".
- An organisational system with multiple levels, including modules, classes, functions, and packages. These help with the organisation of code.
- Compile on the fly to byte code: Without performing a separate compilation process, source code is directly compiled to byte code. It is also possible to "pre-compile"source code modules into byte code files.
- Object-oriented: Python provides a consistent way to use objects: everything is an object.
- Extensions in C and C++ - Extension types and modules can be manually written.

Definitions and statements can be stored in Python files and then used in scripts or interactive interpreters. A module is the name for such a file. It is also possible to import pre-designed modules into Python. Modules make up a library.

2.3 Python's libraries

In this section, it will be explored some of Python's libraries that are crucial for various parallel programming tasks. These libraries (Python's multiprocessing,

Python's futures, Torc_Py, SCOOP, and Dask) offer powerful tools for parallel processing, concurrent execution, and distributed computing.

2.3.1 Python's multiprocessing

The Python multiprocessing module is a library for writing concurrent [15], parallel code using the Python programming language. It provides a high-level interface for creating and managing processes, as well as several tools for synchronisation, communication and process pool management. The module allows for the distribution of workloads across multiple CPU cores, helping to improve performance and speed up execution time.

Although this is a well-built and well-managed module, it is not a suitable choice for distributed memory systems; even though it can spawn multiple processes, they will be bound to a single node, defeating the purpose of the distributed memory. We are looking for a framework that can handle the spawning of processes across multiple nodes, providing the respective mechanism for communication between the processors.

2.3.2 Python's futures

Python's concurrent futures is a library for executing functions asynchronously and concurrently [16]. It provides a high-level interface for managing and executing asynchronous function calls, as well as several classes and functions for launching parallel tasks and managing execution results. This module is built on top of the multiprocessing module and is particularly useful for parallelising the execution of independent, I/O-bound tasks, such as web scraping, data processing, and similar operations.

The concurrent futures module in Python provides essential support for executing functions asynchronously and concurrently, but it is not explicitly designed for distributed computing across multiple nodes in a cluster. While it is possible to implement a custom executor class that allows for the execution of tasks on remote nodes, this requires a significant amount of additional work and the use of extra tools and frameworks to manage the underlying cluster infrastructure.

2.3.2.1 PEP3148

PEP 3148, also known as the "Futures" proposal [17], is a Python Enhancement Proposal that introduces a standardised way of writing concurrent and parallel

code in Python. It provides a new module called "concurrent.futures"with a simple and standard way of writing concurrent and parallel code.

The significance of PEP3148 is that it presents a simple, uniform, and consistent approach to writing concurrent and parallel code. This makes it effortless for developers to parallelize their code and for others to comprehend and maintain it. Moreover, it allows for the reuse of code across different concurrency libraries, which enhances the code's maintainability and portability.

PEP3148 provides a Future class that represents the outcome of a computation that may not have been completed yet. The module also offers several functions, such as map() and submit(), to work with the Future class. With these functionalities, developers can easily submit tasks to be executed concurrently, which provides a way to wait for multiple futures to complete.

To summarise, PEP3148 provides a standardized way of writing concurrent and parallel code in Python. It enhances the code's maintainability and portability, allows for code reuse across different concurrency libraries, and enables developers to submit tasks to be executed concurrently.

2.3.3 Torc_Py

Torc_Py is a popular library for distributed parallel computing in Python. [18] It is built on top of the multiprocessing module outlined in PEP3148 and provides a higher-level API for creating and managing parallel processes. The library allows for easy parallel execution of Python code across multiple processors, making it a powerful tool for handling large-scale data processing tasks.

Torc_Py, an open-source tasking library, seeks to provide a parallel computing framework that:

- offers a single approach for expressing and executing task-based parallelism on both shared memory platforms and distributed memory architectures.
- uses MPI internally while remaining transparent to the user and allowing the application-level use of legacy MPI code
- offers simple parallel nested loop and map function support.
- permits task stealing at all parallelism levels.
- exports the above functionalities through a simple and single Python package

Torc_Py is a Python library that facilitates parallel computing tasks. It includes features such as shared memory support and a parallel map function that distributes work across multiple processors. Torc_Py also allows the creation of data-parallel processes for efficient processing of large datasets.

The library provides an easy-to-use API for managing parallel processes, with built-in methods for creating, launching, and monitoring parallel processes. Additionally, it includes utilities for debugging and monitoring parallel processes, which is helpful when working with large-scale data processing tasks.

Developers can customize Torc_Py to meet their specific needs. The library includes built-in options for adjusting the number of parallel processes, the size of shared memory, and the level of inter-process communication, making it highly customizable.

Torc_Py is scalable and can handle larger datasets by adding more processors to the system.

In summary, Torc_Py is a valuable tool for parallel computing tasks. Its support for shared memory, data-parallel processes, and customizable options make it easy to execute Python code in parallel across multiple processors.

Torc_Py implements several of the patterns mentioned in section 2.1:

- Map;
- Fork-Join;
- MapReduce;
- Scatter;
- Gather;

The application of the Stencil pattern does not seem to be available in Torc_Py. Torc_Py is focused on offering a high-level parallel computing programming model and making it simple for programmers to create parallel algorithms.

As an alternative, stencil computations can be implemented utilising the tools offered by Torc_Py and other parallel programming libraries. For instance, Torc_Py's scatter, gather, and map patterns might be used to distribute the data and update the grid's elements parallelly. The final result could then be gathered and combined to produce the final outcome.

In summary, Torc_Py is a powerful library for distributed parallel computing in Python. It is built on top of PEP3148 and provides a higher-level API for creating and managing parallel processes. The library includes support for shared memory

and inter-process communication, large data sets, and an easy-to-use API for managing parallel processes.

2.3.4 SCOOP

SCOOP (Scalable COncurrent Operations in Python) is a parallel computing framework for Python that is based on the concept of distributed tasks. [19] Scoop's main advantage is its simplicity: it requires minimal changes to existing Python code and offers a simple, easy-to-use interface for parallelising computations. Scoop can be used to parallelise a wide range of computations, from simple loops to more complex algorithms.

SCOOP is particularly useful for large-scale data processing and simulations, as well as data mining applications, where parallel processing can significantly speed up computations.

SCOOP implements several of the patterns mentioned in section 2.1:

- Map;
- Fork-Join;
- MapReduce;

SCOOP aims to offer a simpler alternative designed for HPC (High-Performance Computing) usage while staying usable on development systems. As of now, it uses ZeroMQ to communicate. ZeroMQ looks like an embeddable networking library but acts like a concurrency framework. It gives sockets that carry atomic messages across various transports like in-process, inter-process, TCP, and multicast. [20]

SCOOP has many features and advantages over Futures, multiprocessing and similar modules, such as:

- Harness the power of multiple computers over the network;
- Ability to spawn subtasks within tasks;
- API compatible with PEP 3148;
- Parallelising serial programs with only minor modifications;
- Efficient load-balancing.

One of Scoop's key features is its ability to parallelise computations using the "map"function. This function is similar to the built-in "map"function, but it applies the provided function to the input in parallel. The function returns an iterator

that yields the results as they become available. This feature allows for simple and efficient parallelisation of computations such as data processing or numerical simulations.

Scoop provides:

- Tasks of different nature or complexity can be executed simultaneously while handling physical considerations of parallelization, such as task distribution and communication.
- A way to cancel a computation. This can be useful in cases where a computation takes too long or when the results are no longer needed.
- A way to add callbacks to the results of a computation. This can be useful to perform some action or update some variables once the computation is done.
- A way to limit the number of concurrent workers. This can be useful in cases where there are not enough resources or to avoid overloading the system.

In summary, Scoop is a parallel computing framework for Python based on the concept of "worker pools". It provides a simple, easy-to-use interface for parallelising computations.

2.3.5 Dask

Dask is a parallel computing library for analytics in Python [21]. It provides an easy-to-use API for executing parallel processing jobs and supports distributed computing across multiple nodes in a cluster. Dask is designed to scale from a single machine to a large cluster and provides a flexible and convenient way to execute parallel processing jobs on large datasets.

Dask is a Python library that offers a simple API similar to other standard libraries like NumPy [22] and Pandas [23]. Dask also provides support for executing parallel processing jobs across multiple nodes in a cluster. With Dask, you can efficiently parallelise your existing Python code and run complex parallel processing jobs.

Regarding distributed computing in clusters, Dask offers a flexible and scalable way to execute parallel processing jobs across multiple nodes. Dask supports multiple backends for executing parallel processing jobs, including the use of distributed schedulers like Dask Distributed.

Dask offers a straightforward communication approach with distributed computing platforms like Hadoop. Hadoop is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models [24]. By doing so, you can use the strength of these systems to run parallel processing tasks on massive datasets.

Additionally, Dask offers tools for tracking and analysing the execution of your code, as well as tools for controlling and monitoring the performance of your parallel processing jobs. This enables you to enhance the scalability and efficiency of your code and maximise the performance of your parallel processing processes.

When datasets outgrow the available Memory, Dask was created to natively scale these packages and the surrounding ecosystem to multi-core machines and distributed clusters [21].

Dask implements several of the patterns mentioned in section 2.1:

- Map;
- Fork-Join;
- MapReduce;
- Gather;
- Scatter;
- Stencil;

The scatter and stencil patterns can be implemented in a scalable and efficient manner using Dask's distributed arrays [25], making it suitable for large-scale data processing and analysis.

2.4 Proposed approach

Torc_Py, SCOOP, and Dask use MapReduce, Fork-Join, and map patterns, which indicates that these libraries have much in common. However, each of these libraries works best for different purposes. Torc_Py excels at parallel computation in high-performance computing (HPC) settings where MPI is frequently employed. It is designed to make use of MPI, offering efficiency and scalability for parallel computing in HPC systems.

SCOOP and Dask are the best options for parallel computing on clusters and multi-core devices when MPI is unavailable. SCOOP offers a high-level API for

parallel programming while hiding the user from the specifics of MPI communication. Dask is a strong option for data analysis and data science applications since it offers parallel computing through the use of a task scheduler and network communication layer.

Considering the distinctive characteristics, the optimal option will ultimately depend on the user's requirements, such as the system type, the kind of parallel computing desired, and, of course, the degree of process control needed.

After considering the available options, it is concluded that Torc_Py is the best choice for this research objectives. This is because it supports the "scatter"and "gather"patterns and relies solely on MPI, which makes it lightweight and minimizes dependencies on additional modules and systems. Torc_Py's implementation simplicity, flexibility, and ease of integration into existing systems make it an ideal choice for extending parallel distributed programming capabilities. Additionally, its scalability and minimal overhead make it a compelling option.

3

Implementation

This chapter explores the implementation of two patterns - the stencil filter pattern and the searching pattern with the ability to abort - in pursuit of our research objectives. Our aim is to enhance the computational functionalities of the Torc_Py Python library, enabling more efficient data processing and analysis across distributed computing environments. Our work addresses the absence of these patterns in Torc_Py documentation and implementations of Torc_Py, thereby expanding the library's usefulness and applicability in real-world scenarios. With careful design and implementation, we strive to equip Torc_Py with advanced capabilities, empowering users to manage and execute parallel distributed computing tasks with greater flexibility and control.

3.1 API Documentation

This section provides documentation for both the `stencil` function and the `search` function, which is utilized for stencil operations and searching specific values in data, respectively.

Stencil Function

```
stencil2D(data: ndarray, radius: int, function: Callable) -> ndarray
```

Parameters

- **data** (`ndarray`): The input 2D matrix on which the stencil operation is to be applied.
- **radius** (`int`): The radius of the filter matrix. It specifies the size of the neighbourhood region around each element in the input matrix.

- **function** (`Callable`): A user-defined function that applies the filter to the input data after receiving a matrix as input that matches the size of the filter matrix. The function returns a single value representing the result of the stencil operation.

Return Value

- **Return Type**: `ndarray`
- **Description**: The function returns a 2D matrix representing the result of applying the stencil operation with the specified filter function on the input data.

Example Usage

```
import numpy as np

# Define the filter function (Example: Left Sum Filter)
def filter_function(sub_array):
    filter = np.array([[0, 0, 0], [1, 1, 0], [0, 0, 0]])
    result = np.sum(sub_array * filter)
    return result

# Generate sample input data
data = np.ones((4, 4))

# Apply stencil operation with radius = 1 and the filter function
result = stencil2D(data, radius=1, function=filter_function)

print("Result after stencil operation:", result)

Result:
```

$$\begin{bmatrix} 1 & 2 & 2 & 2 \\ 1 & 2 & 2 & 2 \\ 1 & 2 & 2 & 2 \\ 1 & 2 & 2 & 2 \end{bmatrix}$$

This code snippet demonstrates the application of the `stencil` function with a user-defined filter function (`filter_function`) on a 4x4 matrix (`data`) consisting

of all ones. The resulting value represents the outcome of the stencil operation. Adjust the parameters and filter function as per your requirements.

Search Function

```
search(arr: List, checkFunction: Callable, transformFunc: Callable)
    -> Tuple[bool, Optional[int], Any, Any]
```

Parameters

- **arr** (`List`): The input array of data in which the search operation is to be performed.
- **checkFunction** (`Callable`): A user-defined function that accepts a value from the input array as input and returns `True` if the value meets the search criteria, and `False` otherwise.
- **transformFunc** (`Callable`): A user-defined function that accepts a value from the input array as input and performs a transformation operation on it. This transformation may alter the value's representation in the array.

Return Value

- **Return Type**: `Tuple[bool, Optional[int], Any, Any]`
- **Description**: The function returns a tuple containing the following elements:
  - `found` (`bool`): A boolean flag indicating whether the search criteria were met (`True`) or not (`False`).
  - `index` (`Optional[int]`): The index where the value was found. If the value was not found, `index` is `None`.
  - `originalValue` (`Any`): The original value in the array where the search criteria were met.
  - `transformedValue` (`Any`): The value after being processed by the `transformFunc`.

Example Usage

```
# Define the check function
def check_function(value):
    return value == 42
```

```
# Define the transformation function
def transform_function(value):
    return value * 2

# Generate sample input array
arr = [1, 2, 3, ..., 100]  # Assuming an array of integers from 1 to 100

# Perform the search operation
found, index, original_value, transformed_value = search(
    arr, checkFunction=check_function, transformFunc=transform_function
)

if found:
    print("The Value found at index {} with original value {} and transformed
    value {}.".format(
        original_value, index, transformed_value
    ))
else:
    print("Value not found within the array.")


Result:
The value found at index 20 with original value 21 and transformed value 42.
```

This code snippet demonstrates the application of the `search` function on an input array `arr`. The `check_function` is defined to check if a value equals 42, and the `transform_function` doubles the value. Adjust the parameters and functions as per your requirements.

3.2 Search Pattern

The search pattern that was implemented for this research in the Torc_Py library facilitates parallel searching operations across distributed systems. It leverages parallel processing capabilities to expedite the search for specific values within arrays. The following aspects are highlighted within the implementation:

- **Functionality**: The search pattern initiates the search operation by distributing the workload among available processing units. It coordinates the

execution of search tasks and aggregates results to efficiently determine the presence of target values.

- **Parallel Processing**: The implementation employs parallel processing techniques to enhance the search performance. Individual worker nodes execute search operations on assigned segments of the input array, utilizing collaborative communication to expedite the detection process.

- **Communication Mechanisms**: The search pattern incorporates broadcasting mechanisms to notify other processors upon detecting target values. This collaborative communication ensures coordinated termination of processing tasks across multiple nodes, enhancing overall efficiency.

3.3 Stencil Filter Pattern

The stencil filter pattern that was implemented for this research extends the computational capabilities of the Torc_Py library by facilitating stencil operations on multidimensional data. It applies user-defined filter functions to neighbourhood data regions (halo), enabling various processing tasks such as blurring, edge detection, and image processing. The implementation of the stencil filter pattern encompasses the following key components:

- **Stencil Operation Function**: The `Stencil Operation` function orchestrates the stencil computation process, handling data partitioning between neighbouring nodes and applying the specified filter function to compute results efficiently.

- **Parallel Processing**: The stencil filter pattern utilizes parallel processing techniques to distribute computational tasks across multiple nodes. Each worker node performs stencil computations on assigned input data segments, enhancing processing throughput and scalability.

- **Communication and Data Exchange**: The implementation incorporates communication primitives, such as MPI send and receive operations, to exchange data between neighbouring nodes during stencil computations. This collaborative data exchange mechanism enables seamless integration of stencil operations within distributed computing environments.

3.4 Integration and Workflow

Integrating both the search pattern and the stencil filter pattern into the Torc_Py library provides a comprehensive framework for distributed data processing. The combined functionality allows users to efficiently perform complex analytical tasks across distributed systems. The workflow involves distributing computational tasks, executing parallel operations, and aggregating results, which facilitates a seamless integration of parallel computing techniques into data analysis workflows.

During the implementation process, the data is divided equally among the distributed processes, ensuring balanced workload distribution and optimal resource utilization. Specifically, in the stencil pattern, the matrix rows are evenly divided among the worker processes, which facilitates parallel computation and efficient processing of large datasets.

MPI's scatter and gather functions distribute and collect data across the distributed computing environment, respectively. The scatter function distributes dataset segments to each worker process, while the gather function collects the results from all processes and consolidates them into a cohesive output.

Furthermore, communication between processes is facilitated through MPI's *Isend* and *Recv* functions, which allow asynchronous data sending and receiving. This approach enhances performance by enabling processes to continue computation while data transfer operations are in progress.

Additionally, the MPI Iprobe function is used to test for message arrival in a non-blocking manner. This feature is particularly useful in the search pattern, where it allows processes to check if another worker has discovered a specific value and subsequently broadcast this information to all other workers. Non-blocking communication techniques enable the search pattern to efficiently coordinate the actions of distributed processes without introducing unnecessary delays or bottlenecks.

3.5 Conclusion

Implementing the search and stencil filter patterns within the Torc_Py library significantly advances distributed data processing capabilities. By leveraging parallel processing techniques and collaborative communication mechanisms, the patterns empower users to perform complex computations efficiently across distributed computing environments. The next section will delve into experimental

evaluations and performance analyses to assess the effectiveness and scalability of the implemented patterns.

The details of the implementation can be found here: Online Git Repository

4

Evaluation

This chapter assesses the performance and scalability of the stencil and search patterns implemented through comprehensive experimentation and analysis as also the simplicity for the programmer. The evaluation aims to validate the system's effectiveness, efficiency, and scalability in addressing the objectives outlined in the earlier chapters. It examines the system's behaviour concerning various parameters such as time complexity, execution time, and scalability across different processor configurations.

4.1 Experimental Setup

Before delving into the results, it is essential to detail the experimental setup utilised in the evaluation. This includes the hardware and software configurations employed to execute the experiments, ensuring the reliability of the results. Furthermore, the metrics considered for evaluation, including time utilisation on a single processor, execution on multiple processors, and the impact of the number of processors, are explicitly defined.

4.1.1 Hardware Configuration

To benchmark the performance of the implemented patterns, it was used 3 "Charmander" nodes of the Department of Informatics (DI) cluster at NOVA School of Science and Technology (NOVA FCT). Each node is composed of:

- **CPU**: AMD EPYC 7281 with 16 cores and 32 threads and a base clock speed of 2.1GHz
- **Memory**: 128 GiB DDR4 2666 MHz
- **Network Interface**: 2 x 10 Gbps

- **Main Disk**: 1.8 TB HDD

4.1.2 Software Configuration

The patterns were implemented and evaluated using a software environment primarily based on Python 3.9.2. It was used the version 0.1.1 of the Torc_Py library. Moreover, the Python library "mpi4py"supports various functionalities. This library was used to enable parallel computing patterns and communication among distributed processes.

The "mpi4py" library provides essential tools and utilities for parallel computing. It efficiently utilises the hardware resources available on the "Charmander" nodes of the DI cluster at NOVA School of Science and Technology (NOVAFCT).

4.2 Performance Analysis

With the experimental setup in place, the stencil's performance and search patterns implemented under various conditions will be analysed. Each experiment is designed to assess the system's behaviour concerning the defined metrics and parameters.

4.2.1 Experimental Metrics

In order to evaluate the performance and scalability of the implementation of patterns, it is crucial to consider certain metrics.

Defining the variables listed below:

- T_1: Execution time on a single processor;
- T_p: Execution time on p processor;
- p: Number of processors.
- S_p: Speedup;
- E_p: Efficiency;
- C_p: Cost;

Using these formulas to relate them to one another:

$$S_p = \frac{T_1}{T_p}, \quad E_p = \frac{S_p}{p}, \quad C_p = p \cdot T_p$$

The following observations can be drawn from these definitions:

- **Speedup** calculates the speed at which an execution utilizing p processors is faster than one using a single processor;
- **Efficiency** Shows the average speedup we receive for each processor;
- **Cost** calculates the overall execution time of a p processor execution.

A parallel algorithm is cost-optimal when $C_p = T_1$ for any p. This means that the added parallelism does not introduce undesired overheads that would raise the total amount of work required in contrast to serial execution. In this scenario, parallel efficiency is perfect ($E_p = 1$), given that all work would be divided equally across processors.

Even though this scenario is perfect, it is rarely encountered in practical applications since efficiency is nearly always less than 100% due to inefficient task distribution or communication overheads. However, the ideas of perfect parallel efficiency and cost-optimality are still relevant since they serve as a reminder that parallel algorithms should work to minimise their overheads in order to maximise the possible speedup.

4.2.2 Performance and Scalability Analysis

The time utilisation on a single processor serves as a baseline metric to evaluate the system's algorithmic implementation efficiency. More about the system's underlying computational complexity and optimisation potential is learned by evaluating the execution time on a single processor.

Experiments on multiple processors are conducted to evaluate the system's parallelizability, assessing the system's ability to exploit parallelism for enhanced performance. By comparing execution times across different processor configurations, it is possible to analyse the scalability and efficiency of parallel execution.

Scalability analysis is crucial for determining how the system's performance scales with an increasing number of processors. It investigates the system's behaviour under varying computational loads, examining its ability to maintain efficiency and performance as the workload distribution changes.

4.2.3 Comparison with Existing Patterns

During the evaluation process, new patterns (*search* and *stencil*) were added to the Torc_Py library and compared with the existing ones within and outside the library. The aim of the comparison was to understand the advantages and disadvantages

of incorporating new patterns into the library versus sticking with the available methods.

To assess the new patterns' performance, experiments were conducted using both Torc_Py's MPI and legacy MPI. This allowed to analyze the speed, scalability, and efficiency of the new patterns compared to the existing methods.

The comparison involved looking at factors like execution time, speedup, efficiency and cost metrics for both sets of implementations. Analysing these metrics makes it possible to see the strengths and weaknesses of introducing patterns into the library versus utilising existing methods.

This analysis can help decide whether it would be beneficial to integrate patterns into the library, considering factors like performance trade-offs and the ease of use and programming. It allows to weigh whether investing time and effort in implementing patterns would be worthwhile.

4.3 Testing Scenarios

4.3.1 Search Pattern

This pattern evaluates the performance of an optimised search pattern designed to locate a specific element within an extensive array of *160 million indexes*. Unlike conventional search patterns, this optimised pattern incorporates the ability to inform other processors to stop their work as soon as one processor finds the desired value.

To realistically simulate the computational workload, a simulated work delay of 0.01 ms is introduced between each iteration in the search process. This time interval was carefully chosen to strike a balance between computational accuracy and efficiency.

It's worth noting that selecting an appropriate simulation time is critical, especially when dealing with large datasets. For instance, iterating through all indexes without delay in an array of 160 million elements would lead to an unrealistic computational load where overheads can overtake the parallel improvements. Also, looking at examples of some Direct Search Optimization Algorithms, the function to optimize usually takes some time, depending on the problem domain and algorithm. To illustrate, if the simulated work time were 1 ms per iteration, it would take approximately 44 hours with a single processor to complete the search. By reducing the simulated work time to 0.01 ms per iteration, the total execution time is significantly reduced to approximately 30 to 40 minutes, making the simulation more practical and feasible.

4.3.1.1 Search for the First Index

This test assesses the search algorithm's efficiency in handling large datasets and provides insights into the system's ability to locate the first occurrence of an element within the array. The tables with the calculated metrics are detailed in Annex I

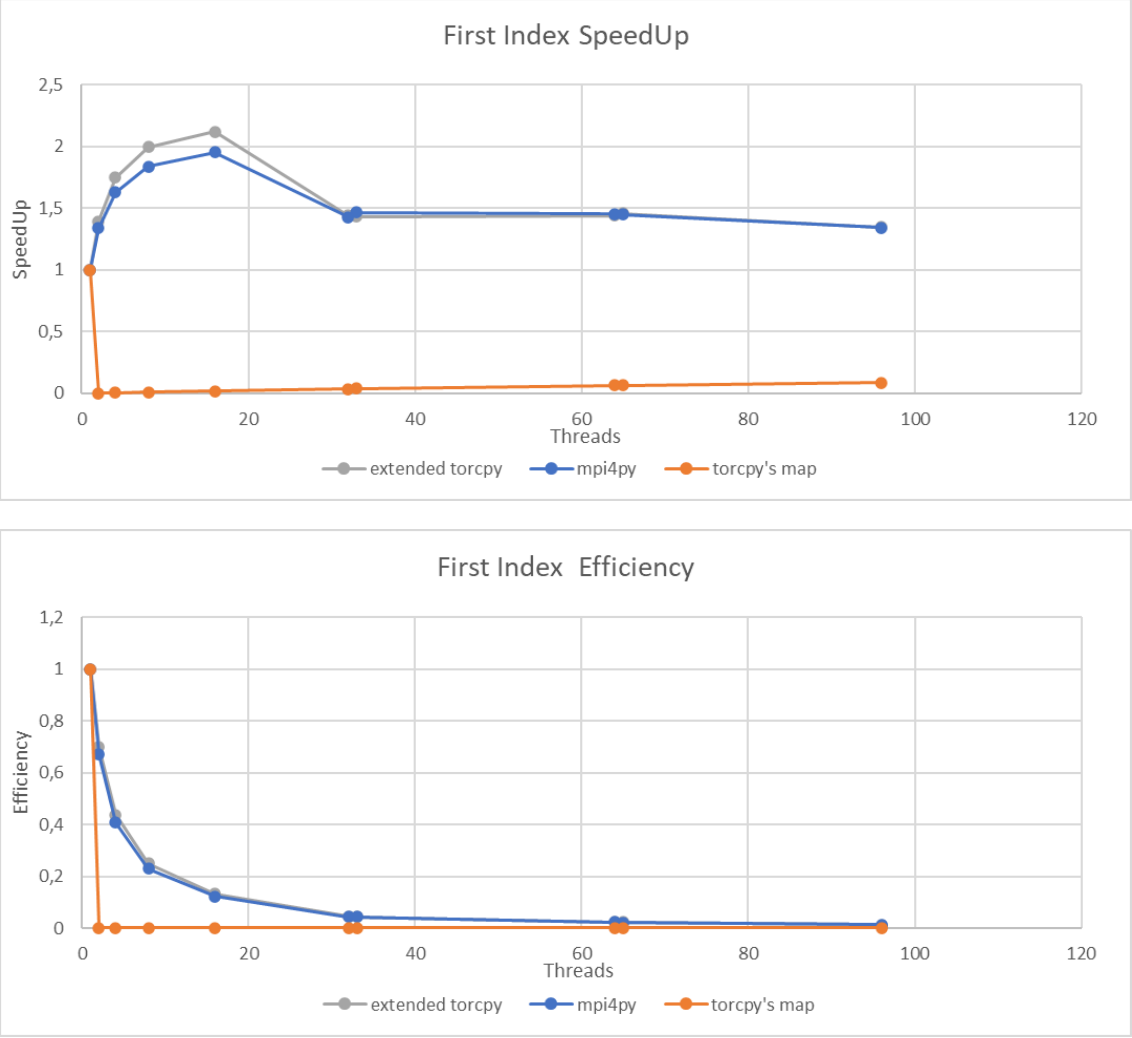


Figure 4.1: Search First Index Metrics

Based on the experimental results and calculated metrics for the "Search for the First Element" test scenario, the following conclusions can be drawn:

1. The default MPI implementation (mpi4py) and the extended torcpy perform significantly better than the Torc_Py map pattern. The default mpi and the extended Torc_Py implementation reduce execution times and improve metrics like speedup, efficiency, and cost. This indicates that Torc_Py's mapping functionality lacks the ability to inform other processes about the discovered values.

2. The difference in time between Torc_Py's map and other patterns is that the map version lacks the ability to inform other processes about the discovered values. As a result, the other processes cannot abort their work. In a scenario with two workers, the first worker would finish almost immediately, while the second worker would need to iterate for their half of the array, resulting in a significant increase in time spent.

3. After analyzing the metrics, it has been found that both the default MPI pattern and the custom Torc_Py extension perform better in terms of scalability and resource utilization as compared to the Torc_Py's map pattern. This indicates that the parallel processing capabilities are quite effective, and it is possible to inform other processes that a value has been found, which leads to other processes stopping the search for the first element within the array, thereby optimizing the search process.

The following findings highlight the significance of choosing suitable parallel processing frameworks and patterns that match particular search tasks. In this situation, both the default MPI pattern and the extended Torc_py are the recommended option because of its excellent performance and efficiency comparing to the Torc_Py's map pattern. However, to fully leverage the potential of custom extensions in enhancing parallel processing tasks related to finding the first element, further refinement and optimization might be necessary.

4.3.1.2 Search for the Last Element

This test evaluates the effectiveness of the search algorithm in identifying the last occurrence of an element within the array. It highlights any potential optimisations or bottlenecks in the search process. The tables with the calculated metrics are detailed in Annex II

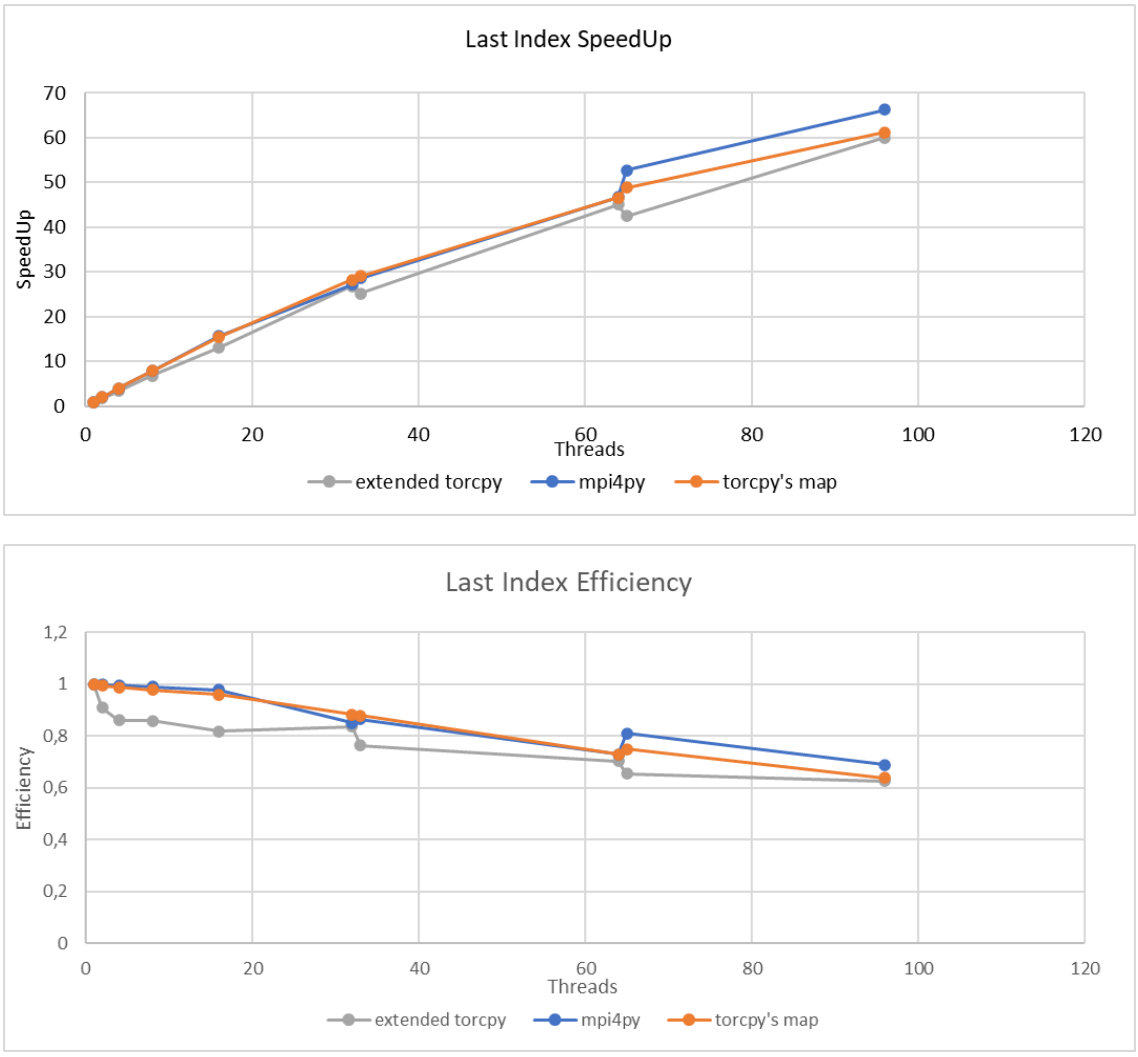


Figure 4.2: Search Last Index Metrics

Based on the experimental results and calculated metrics for the "Search for the Last Element" test scenario, the following conclusions can be drawn:

1. The program's performance using the Torc_Py map pattern is slightly better than the program using only the default MPI implementation (mpi4py). This improvement is evident in terms of reduced execution times and better-calculated metrics, including speedup, efficiency, and cost.

2. Interestingly, the custom Torc_Py extension (extended torcpy) performs slightly worse than both the Torc_Py map pattern and the default MPI implementation in this testing scenario. Despite potential optimisations or functionalities offered by the custom extension, its performance does not match the efficiency of the other implementations for searching the last element in the array.

3. Analysis of the calculated metrics reveals that the Torc_Py map pattern exhibits better scalability and resource utilisation than the default MPI implementation and the custom Torc_Py extension. This suggests that using Torc_Py's mapping functionality contributes to more efficient parallel processing and better overall performance in this particular search scenario.

These conclusions highlight the importance of considering the specific characteristics of the search task and the underlying parallel processing patterns when choosing the most appropriate implementation. While the TorcPy map pattern demonstrates advantages over the default MPI implementation in this scenario, further optimisation or refinement may be needed for custom extensions to realise their full potential.

4.3.1.3 Search for the Middle Element

This test aims to demonstrate the effectiveness of the optimised search pattern in reducing overall execution time by minimising unnecessary computations across multiple processors. The optimised pattern enhances efficiency and scalability by leveraging early termination, particularly in scenarios where the target element is found relatively early in the search process. The tables with the calculated metrics are detailed in Annex III

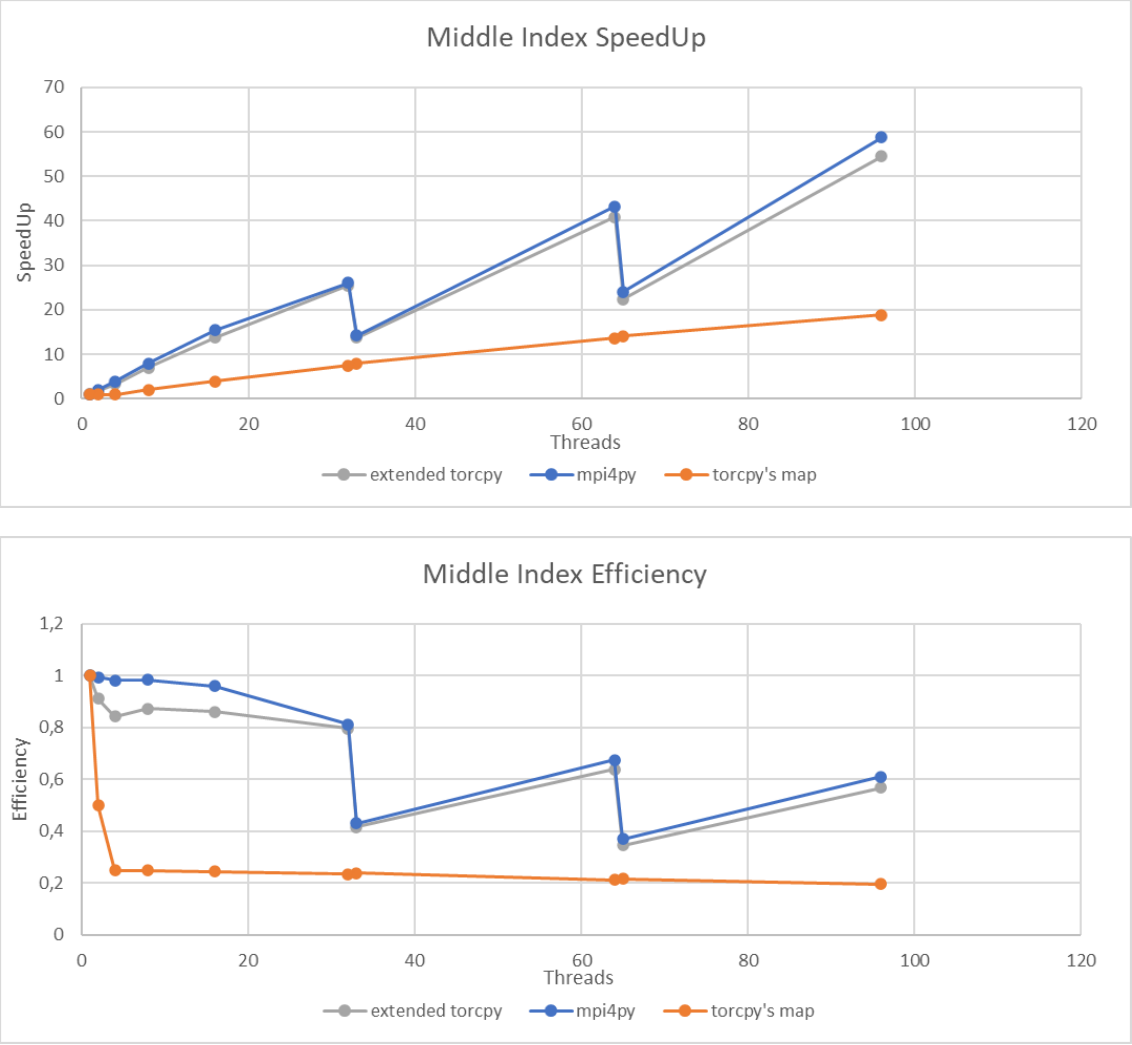


Figure 4.3: Search Middle Index Metrics

Based on the experimental results and calculated metrics for the "Search for the Middle Element" test scenario, the following conclusions can be drawn:

1. The performance of the program using the Torc_Py library with the extended search pattern (extended torcpy) surpasses the program using the Torc_Py map pattern in terms of reduced execution times and improved calculated metrics, including speedup, efficiency, and cost. This suggests that the extended search pattern offers optimisations that are beneficial for efficiently finding the middle element in the array.

2. The reason for the time difference between Torc_Py's map and other patterns is the same as explained before in 4.3.1.1. In a scenario where there are two or more workers, one worker would finish their work by finding the value in the middle of their assigned array, while the other workers would need to iterate through all of their designated arrays. This would result in a significant increase in the time taken to complete the task.

3. Despite the slight advantage of the Torc_Py-based implementations, the default MPI version (mpi4py) exhibits slightly better performance in this testing scenario. This difference could be attributed to the underlying MPI implementation used in each case, with the Torc_Py-based version leveraging its version of MPI while the default MPI version operates independently.

4. While the extended Torc_Py pattern demonstrates better scalability and resource utilisation compared to the Torc_Py map pattern in terms of speedup and efficiency, it is noteworthy that the default MPI version outperforms both Torc_Py patterns in terms of cost. This indicates that the default MPI implementation may be more cost-effective for this particular search task despite slightly lower speedup and efficiency metrics.

5. Additionally, it is worth noting that implementing the search pattern using the extended features of Torc_Py should be more user-friendly and intuitive than directly utilizing MPI. This observation highlights the advantage of leveraging higher-level abstractions provided by Torc_Py, which abstract away much of the complexity associated with manual MPI programming. Consequently, developers may find it easier and more straightforward to program complex parallel distributed tasks using the predefined patterns offered by Torc_Py, leading to increased productivity and reduced development time.

These conclusions underscore the importance of considering the specific characteristics of the search task and the underlying parallel processing patterns when choosing the most appropriate implementation. Further investigation into the differences in performance and optimisations between the Torc_Py map pattern and the extended Torc_Py pattern is warranted to comprehensively understand

their respective strengths and weaknesses.

4.3.2 Stencil Pattern

The stencil pattern is commonly used in image processing, computational fluid dynamics, and other scientific computing applications for tasks such as smoothing, edge detection, and convolution.

This evaluation examines the efficiency and scalability of the stencil pattern implementation across different processor configurations. Performance metrics such as execution time, speedup, efficiency, and cost to analyse the impact of parallelisation on overall system performance.

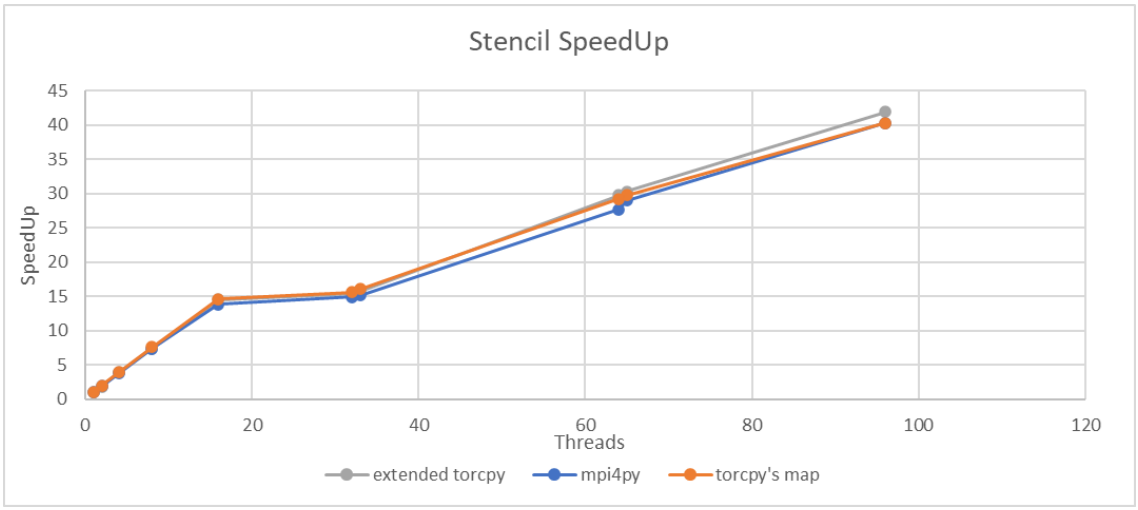
Evaluating the stencil pattern implementation aims to gain insights into its suitability for parallel processing tasks and its potential benefits for computational efficiency in large-scale scientific computing applications.

4.3.2.1 Stencil Pattern with Filter: Left sum filter

This pattern assesses the performance of a stencil pattern applied to a matrix of size $10,000 \times 10,000$ composed entirely of 1's. The stencil pattern involves applying a filter matrix to each element of the input matrix, where the filter matrix is defined as:

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

This test evaluates the system's performance by applying the stencil pattern with the given filter across the array. The tables with the calculated metrics are detailed in Annex IV



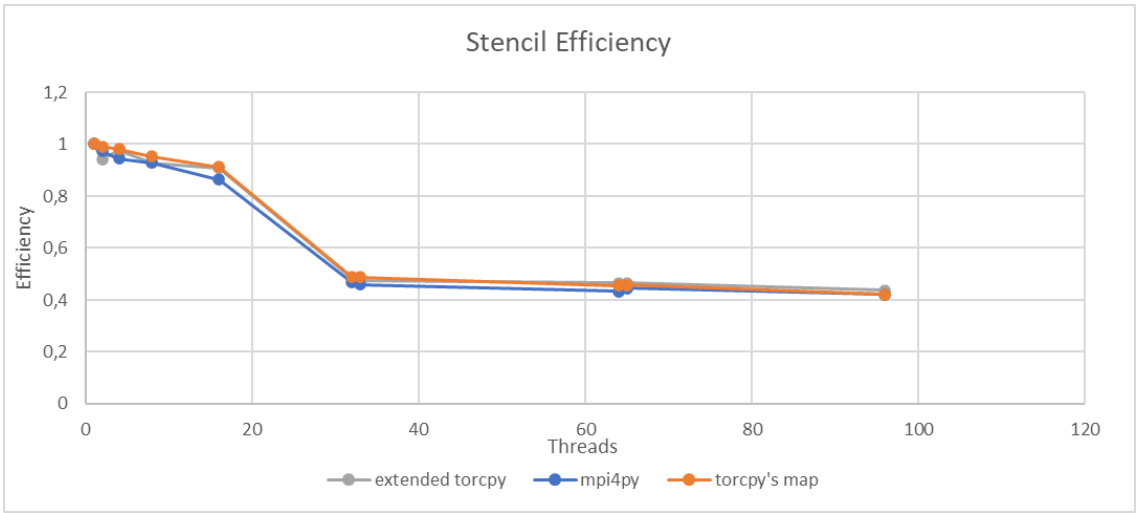


Figure 4.4: Stencil Metrics

Based on the experimental results and calculated metrics for the "Stencil Pattern with Filter: Left sum filter" test scenario, the following conclusions can be drawn:

1. The performance of the program using the Torc_Py library with the extended stencil pattern (extended torcpy) is comparable to the program using the Torc_Py map pattern in terms of execution time and calculated metrics, including speedup, efficiency, and cost. This suggests that the extended stencil pattern offers optimisations beneficial for applying the left-sum filter across the array.

2. The Torc_Py map pattern exhibits a small performance loss compared to the previous tests of the search pattern (as in the search pattern using the map pattern would always have a slightly better performance compared to the other tests). This loss in performance may be attributed to the communication overhead required for the map pattern to achieve the same functionality as the different patterns, such as dividing the matrix and exchanging neighbour parts of subarrays.

3. Both Torc_Py-based implementations (torcpy's map and extended torcpy) slightly outperform the default MPI version (mpi4py) regarding speedup and efficiency. However, the default MPI version exhibits a lower cost compared to the Torc_Py-based implementations in certain scenarios (using 1, 2, or 8 threads), indicating that it may be more cost-effective for this particular scenario.

4. The extended stencil pattern demonstrates the scalability and resource utilisation comparable to the Torc_Py map pattern, suggesting that it provides efficient parallel processing capabilities for applying the left-sum filter across the array. However, further investigation is warranted to understand the trade-offs and optimisations involved in each implementation.

These conclusions underscore the importance of considering the specific characteristics of the stencil pattern task and the underlying parallel processing patterns when choosing the most appropriate implementation. Further optimisation and fine-tuning of the Torc_Py-based implementations may improve performance and resource utilisation, offering better scalability and efficiency in stencil pattern applications.

5

Conclusions

A comprehensive study was carried out on various Python libraries for parallel computing. There was an evaluation of Torc_Py's library performance and suitability for different use cases. The extended version of Torc_Py was the primary focus, and it incorporated two new patterns into its API. The implementation and performance enhancements of these patterns were assessed.

The study's objectives were to examine the current landscape of parallel programming libraries, extend Torc_Py's API to accommodate additional patterns while ensuring compatibility with existing functionalities, and thoroughly evaluate the extended library's performance.

The analysis revealed that the extended Torc_Py library shows significant performance enhancements in specific scenarios (search pattern where the element is in the middle of an array). However, creating and integrating custom patterns into the library may pose challenges regarding time and resource allocation. Nonetheless, the extended Torc_Py library remains a valuable asset for users seeking to address specialized parallel computing tasks beyond the capabilities of existing methods or the default MPI version.

Furthermore, the default MPI implementation demonstrated lower costs than Torc_Py-Based solutions, particularly under specific thread configurations. This underscores the importance of considering performance and cost factors when selecting the appropriate parallel computing framework.

In addition to performance considerations, the potential benefits of Torc_Py's ease of programming through predefined patterns were recognized. Especially in scenarios requiring complex parallel patterns, such as search algorithms on large datasets, Torc_Py's ability to facilitate early termination and dynamic workload distribution proved advantageous, leading to significant reductions in execution time and enhanced efficiency.

After analyzing the data, it's evident that the default MPI version performs

the best in almost every scenario. On the other hand, Torc_Py's map performs the worst in the first index search and the middle index search scenarios. Although the default MPI version is the best performer, extended Torc_Py performs almost as well. For a programmer, using an already designed and implemented pattern is easier. Hence, it can be concluded that the extended Torc_Py is the best option for a programmer.

Future research and development should prioritize mitigating the overhead associated with custom pattern creation in Torc_Py, enhancing its accessibility and user-friendliness. Furthermore, continued optimization and refinement of Torc_Py's underlying mechanisms are essential for improving its performance and competitiveness within the parallel computing landscape.

In conclusion, while the extended version of Torc_Py exhibits promise in enhancing parallel computing performance, its adoption should be carefully considered in light of associated costs and complexities. Users intending to develop custom patterns using the default MPI may find leveraging Torc_Py's features beneficial in enhancing performance, particularly in tasks where dynamic workload distribution and early termination are advantageous. By carefully evaluating task requirements and considering factors such as performance, cost, and ease of implementation, users can make informed decisions regarding the most suitable parallel computing solution for their needs. Ultimately, the aim is to balance performance gains and practical considerations, ensuring efficient and effective utilization of parallel computing resources.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. ii).

[2] A. D. M. P. M. d. Santos. *Reimplementation of the SID-PSM Derivative-Free Optimization Algorithm in Python*. 2023-12. URL: https://run.unl.pt/handle/10362/164325 (cit. on p. 2).

[3] L. M. F. F. Monteiro. *Python vs Julia vs Matlab para programação*. 2023-02. URL: https://run.unl.pt/handle/10362/144294 (cit. on p. 2).

[4] A. I. F. V. A. L. Custódio J. F. A. Madeira and L. N. Vicente. "Direct Multisearch for Multiobjective Optimization". In: *SIAM Journal on Optimization. 21(3)*. 2011, pp. 537–555. DOI: https://doi.org/10.1137/050646706 (cit. on p. 3).

[5] A. L. Custódio and L. N. Vicente. "Using Sampling and Simplex Derivatives in Pattern Search Methods". In: *SIAM Journal on Optimization 18.2*. 2007, pp. 537–555. DOI: https://doi.org/10.1137/050646706 (cit. on p. 3).

[6] S. T. C. P. B. A. L. C. V. Duarte and P. Medeiros. "Parallel strategies for Direct Multisearch". In: *Numerical Algorithms volume 92*. 2023, pp. 1757–1788. DOI: https://doi.org/10.1007/s11075-022-01364-1 (cit. on p. 3).

[7] A. R. Michael McCool James Reinders. *Structured Parallel Programming*. First. Morgan Kaufmann, 2012. ISBN: 978-0-12-415993-8 (cit. on pp. 6, 8–10).

[8] *A gentle introduction to parallel patterns*. URL: https://mats-brorsson.medium.com/the-map-parallel-pattern-e002b1a1b48e (visited on 2024-03-31) (cit. on p. 7).

[9] *Stencil Pattern Stencil Pattern Parallel Computing CIS 410/510 Department of Computer and Information Science*. URL: https://slideplayer.com/slide/7070447/ (visited on 2024-03-31) (cit. on p. 7).

[10] *Convolution Matrix*. URL: https://docs.gimp.org/2.8/en/plug-in-convmatrix.html (visited on 2023-02-09) (cit. on p. 8).

[11] *Parallel Control Patterns: Reduction*. URL: https://slideplayer.com/slide/7518581/ (visited on 2024-07-02) (cit. on p. 9).

[12] *Parallel reduction pattern*. URL: https://www.researchgate.net/figure/Figura-12-Padrao-paralelo-Reduce_fig2_328902726 (visited on 2024-07-02) (cit. on p. 9).

[13] *Serial and parallel implementations of the inclusive scan pattern*. URL: https://www.researchgate.net/figure/11-Serial-and-parallel-implementations-of-the-inclusive-scan-pattern-Image-taken-from_fig46_260226184 (visited on 2024-03-31) (cit. on p. 10).

[14] *Python 3.11.1 documentation*. URL: https://docs.python.org/3/ (visited on 2023-02-05) (cit. on p. 11).

[15] *multiprocessing — Process-based parallelism*. URL: https://docs.python.org/3/library/multiprocessing.html (visited on 2023-02-05) (cit. on p. 12).

[16] *concurrent.futures — Launching parallel tasks*. URL: https://docs.python.org/3/library/concurrent.futures.html (visited on 2023-02-05) (cit. on p. 12).

[17] *PEP 3148 – futures - execute computations asynchronously*. URL: https://peps.python.org/pep-3148/ (visited on 2023-02-05) (cit. on p. 12).

[18] *torcpy: supporting task-based parallelism in Python*. URL: https://github.com/IBM/torc_py (visited on 2023-02-05) (cit. on p. 13).

[19] *SCOOP (Scalable COncurrent Operations in Python)*. URL: https://scoop.readthedocs.io/en/0.7/ (visited on 2023-02-05) (cit. on p. 15).

[20] *ZeroMQ*. URL: https://zeromq.org/ (visited on 2024-03-31) (cit. on p. 15).

[21] *Dask a flexible library for parallel computing in Python*. URL: https://docs.dask.org/en/stable/ (visited on 2023-02-05) (cit. on pp. 16, 17).

[22] *NumPy is the fundamental package for scientific computing in Python*. URL: https://numpy.org/doc/stable/ (visited on 2024-03-31) (cit. on p. 16).

[23] *Pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.* URL: https://pandas.pydata.org/docs/index.html (visited on 2024-03-31) (cit. on p. 16).

[24] *Hadoop*. URL: https://hadoop.apache.org/ (visited on 2024-03-31) (cit. on p. 17).

[25] *Stencil Computations with Numba*. URL: https://examples.dask.org/applications/stencils-with-numba.html (visited on 2023-02-09) (cit. on p. 17).

I

Annex 1 Search First Element Tables

Table I.1: Time taken (in seconds) for each program to finish

Threads	mpi4py	torcpy's map	extended torcpy
96	4.97821188	45.6551187	5.73727107
65	4.60286355	59.90094852	5.296488523
64	4.600907803	61.79236031	5.366460085
33	4.552094698	105.917074	5.390238285
32	4.683664322	112.0543506	5.366804361
16	3.422338247	213.2609763	3.645988703
8	3.636736155	422.4022696	3.872285366
4	4.096531153	840.0672925	4.417888165
2	4.989563704	1674.399514	5.540937185
1	6.681986332	4.012511253	7.731530666

Table I.2: Calculated metrics for mpi4py

Threads	Speedup	Efficiency	Cost
96	1,342246271	0,013981732	477,9083405
65	1,451702024	0,022333877	299,1861308
64	1,452319112	0,022692486	294,4580994
33	1,467892646	0,044481595	150,219125
32	1,42665782	0,044583057	149,8772583
16	1,952462278	0,122028892	54,75741196
8	1,837358018	0,229669752	29,09388924
4	1,631132801	0,4077832	16,38612461
2	1,339192508	0,669596254	9,979127407
1	1	1	6,681986332

Table I.3: Calculated metrics for torcpy's map

Threads	Speedup	Efficiency	Cost
96	0.087887434	0.000915494	4382.891396
65	0.066985772	0.00103055	3893.561654
64	0.064935394	0.001014616	3954.71106
33	0.037883517	0.001147985	3495.263441
32	0.035808616	0.001119019	3585.73922
16	0.018815028	0.001175939	3412.175621
8	0.009499265	0.001187408	3379.218157
4	0.004776416	0.001194104	3360.26917
2	0.002396388	0.001198194	3348.799027
1	1	1	4.012511253

Table I.4: Calculated metrics for extended torcpy

Threads	Speedup	Efficiency	Cost
96	1.347597241	0.014037471	550.7780228
65	1.459746515	0.022457639	344.271754
64	1.440713346	0.022511146	343.4534454
33	1.434357863	0.04346539	177.8778634
32	1.440620926	0.045019404	171.7377396
16	2.120558042	0.132534878	58.33581924
8	1.996632463	0.249579058	30.97828293
4	1.750051241	0.43751281	17.67155266
2	1.395347106	0.697673553	11.08187437
1	1	1	7.731530666

II

Annex 2 Search Last Element Tables

Table II.1: Time taken (in seconds) for each program to finish

Threads	mpi4py	torcpy's map	extended torcpy
96	27.39693427	27.2958765	30.29306102
65	34.44680953	34.26284313	42.79181933
64	38.85771751	35.82782602	40.37365961
33	63.60311484	57.58975482	72.22810245
32	66.67342162	59.20573378	67.88136077
16	116.0175724	108.7554734	139.033752
8	229.0875027	213.3673551	264.7352672
4	455.0539203	422.9141636	528.1188624
2	908.7422223	840.9886758	1001.41415
1	1815.257994	1671.479538	1819.753483

Table II.2: Calculated metrics for mpi4py

Threads	Speedup	Efficiency	Cost
96	66.2577052	0.690184429	2630.10569
65	52.69742014	0.810729541	2239.042619
64	46.71550751	0.729929805	2486.893921
33	28.54039458	0.864860442	2098.90279
32	27.22611125	0.850815976	2133.549492
16	15,6464056	0,97790035	1856,281158
8	7,923863033	0,990482879	1832,700022
4	3,989105276	0,997276319	1820,215681
2	1,997549965	0,998774982	1817,484445
1	1	1	1815,257994

Table II.3: Calculated metrics for torcpy's map

Threads	Speedup	Efficiency	Cost
96	61.23560595	0.637870895	2620.404144
65	48.78402914	0.750523525	2227.084804
64	46.65311082	0.728954856	2292.980865
33	29.02390439	0.879512254	1900.461909
32	28.23171729	0.882241165	1894.583481
16	15,36915326	0,960572078	1740,087574
8	7,83381102	0,979226377	1706,938841
4	3,952290279	0,98807257	1691,656654
2	1,987517295	0,993758647	1681,977352
1	1	1	1671,479538

Table II.4: Calculated metrics for extended torcpy

Threads	Speedup	Efficiency	Cost
96	60.0716277	0.625746122	2908.133858
65	42.52573299	0.654242046	2781.468257
64	45.07279005	0.704262345	2583.914215
33	25.19453539	0.763470769	2383.527381
32	26.80785215	0.83774538	2172.203545
16	13,08857351	0,818035845	2224,540031
8	6,873861208	0,859232651	2117,882137
4	3,445727114	0,861431778	2112,47545
2	1,817183713	0,908591856	2002,8283
1	1	1	1819,753483

III

Annex 3 Search Middle Element Tables

Table III.1: Time taken (in seconds) for each program to finish

Threads	mpi4py	torcpy's map	extended torcpy
96	15,54587436	44,58191133	16,8159368
65	37,91999292	59,70697641	40,83949542
64	21,1434958	61,44652057	22,40461874
33	64,3447361	106,3428144	66,79779387
32	35,10013628	111,5693557	35,93297458
16	59,47102714	213,1758986	66,51792622
8	115,9204593	422,0427125	131,0246336
4	232,3324714	840,7122431	272,0152717
2	458,9187052	838,6278739	502,9394581
1	912,8602414	837,8604488	916,5313566

Table III.2: Calculated metrics for mpi4py

Threads	Speedup	Efficiency	Cost
96	58,72041806	0,611671021	1492,403938
65	24,07332309	0,370358817	2464,79954
64	43,17451807	0,674601845	1353,183731
33	14,18702285	0,429909783	2123,376291
32	26,00731331	0,812728541	1123,204361
16	15,34966328	0,959353955	951,5364342
8	7,874884616	0,984360577	927,3636742
4	3,929111742	0,982277936	929,3298855
2	1,989154574	0,994577287	917,8374104
1	1	1	912,8602414

Table III.3: Calculated metrics for torcpy's map

Threads	Speedup	Efficiency	Cost
96	18,793731	0,195768031	4279,863487
65	14,03287353	0,215890362	3880,953467
64	13,63560444	0,213056319	3932,577316
33	7,878862838	0,238753419	3509,312877
32	7,509772225	0,234680382	3570,219383
16	3,930371372	0,245648211	3410,814377
8	1,985250365	0,248156296	3376,3417
4	0,996607883	0,249151971	3362,848972
2	0,999084904	0,499542452	1677,255748
1	1	1	837,8604488

Table III.4: Calculated metrics for extended torcpy

Threads	Speedup	Efficiency	Cost
96	54,5037346	0,567747235	1614,329933
65	22,44227915	0,345265833	2654,567202
64	40,90814341	0,639189741	1433,895599
33	13,72098244	0,415787347	2204,327198
32	25,50669315	0,797084161	1149,855186
16	13,77871213	0,861169508	1064,286819
8	6,995107188	0,874388398	1048,197069
4	3,369411397	0,842352849	1088,061087
2	1,822349274	0,911174637	1005,878916
1	1	1	916,5313566

IV

Annex 4 Stencil Filter Tables

Table IV.1: Time taken (in seconds) for each program to finish

Threads	mpi4py	torcpy's map	extended torcpy
96	28,18661539	28,56255897	28,06686258
65	39,1885883	38,7173392	38,80396771
64	41,0580562	39,45319017	39,53921843
33	75,05880173	71,73426517	75,05988272
32	76,03007491	73,80728801	76,00418552
16	82,2310009	78,91267053	80,91421088
8	153,127965	151,195253	158,4099698
4	301,0353905	293,4845335	300,8060793
2	584,4576362	581,2395729	623,2054153
1	1136,550503	1152,147331	1175,812873

Table IV.2: Calculated metrics for mpi4py

Threads	Speedup	Efficiency	Cost
96	40,32234758	0,420024454	2705,915077
65	29,00207821	0,446185819	2547,25824
64	27,68154677	0,432524168	2627,715597
33	15,14213492	0,458852573	2476,940457
32	14,94869635	0,467146761	2432,962397
16	13,82143584	0,86383974	1315,696014
8	7,422226913	0,927778364	1225,02372
4	3,775471386	0,943867846	1204,141562
2	1,944624268	0,972312134	1168,915272
1	1	1	1136,550503

Table IV.3: Calculated metrics for torcpy's map

Threads	Speedup	Efficiency	Cost
96	40,33767885	0,420184155	2742,005661
65	29,75791609	0,457814094	2516,627048
64	29,20289401	0,456295219	2525,004171
33	16,06132478	0,486706812	2367,230751
32	15,61021089	0,48781909	2361,833216
16	14,60028311	0,912517695	1262,602728
8	7,620261272	0,952532659	1209,562024
4	3,925751443	0,981437861	1173,938134
2	1,982224516	0,991112258	1162,479146
1	1	1	1152,147331

Table IV.4: Calculated metrics for extended torcpy

Threads	Speedup	Efficiency	Cost
96	41,89327787	0,436388311	2694,418808
65	30,30135686	0,466174721	2522,257901
64	29,7378886	0,464654509	2530,50998
33	15,66499747	0,474696893	2476,97613
32	15,47037002	0,483449063	2432,133937
16	14,53159909	0,908224943	1294,627374
8	7,422593884	0,927824235	1267,279758
4	3,90887337	0,977218343	1203,224317
2	1,886717997	0,943358999	1246,410831
1	1	1	1175,812873