Performance of Parallelism in Python and C++

Francisco Javier Moreno Arboleda, Mateo Rincón Arias, Jesús Antonio Hernández Riveros

Abstract— **In this paper, we evaluate the performance of parallelism in Python and C++. Parallel programming can be achieved in Python through the `multiprocessing` module and in C++ by means of OpenMP directives. For the performance comparison, we use implementations, in parallel and sequential, of three algorithms: frequency (count the occurrences) of an integer in an unsorted array, matrix transposition, and matrix addition. Our goals are i) to show the performance of the algorithms in their parallel and sequential implementations, in Python with `multiprocessing` and in C++ with OpenMP and ii) to show the importance of selecting the programming language and libraries when programming in parallel. Our experiments showed that, in general, C++ outperformed Python. Additional experiments with the naïve matrix multiplication algorithm confirmed this conclusion.**

Index Terms—**Parallel computing, performance, Python, C++, algorithms.**

I. Introduction

THE processing of massive amount of data, ranging from terabytes to petabytes [1], [2] in current applications represents a challenge for organizations [3], [4]. For example, UPS (United Parcel Service) uses massive amount of data and analytics in a number of key projects to improve its performance [5], [6] and thereby stay competitive. However, as more data are generated, the time required to process them can increase to a limit not acceptable to end users.

There are several approaches to deal with this problem, from hardware upgrades, e.g., of processors and networks, to software changes, e.g., redesign of algorithms and database queries, operating systems, programming languages, libraries, development frameworks, and programming paradigms (e.g., to resort to parallel and distributed programming), among others.

In particular, parallel and distributed programming has become popular since 2006 with the emergence of Hadoop and parallel programming techniques such as MapReduce [7]. The decrease both in the cost of processors with shared memory and in the complexity of managing networks of hundreds or thousands of computers have contributed to this boom. From a software point of view, MPI (Message Passing Interface), [8] and OpenMP [9], [10] have been equally instrumental.

Today, parallel programming is used in languages such as C++, Python, Scala, and Rust, among others [11]; in distributed processing frameworks such as Hadoop and Apache Spark; and in DBMS such as Oracle [12], SQL Server [13], [14], and DB2 [15], among others.

In this paper, we evaluate the performance of parallelism in Python and C++. Python is one of the most used programming languages today and through the `multiprocessing` module [16], Python allows us parallel programming. On the other hand, C++ is a programming language widely used for the development of general-purpose programs and scientific computing. By means of OpenMP, we can enable parallel programming in C++.

For the performance comparison we will use implementations, both parallel and sequential, of three algorithms: frequency (count the occurrences) of an integer in an unsorted array, matrix transposition, and matrix addition. We show our experimental setup in Fig. 1 where we show the different implementations for each algorithm. In Fig. 2 we show a flowchart of the activities we followed for the development of our work.

We consider the following hypothesis: in parallel implementations, C++ with OpenMP has lower execution times than Python with `multiprocessing`. Indeed, in performance analysis of *sequential* implementations, it has been reported [17] that when comparing C++ with Python, C++: consumes half the memory, is five to ten times faster for data structure initialization and filling, and requires half the time for indexing the data structures. Our goals are i) to show the performance of the algorithms in their parallel and sequential implementations in Python with `multiprocessing` and in C++ with OpenMP and ii) to show the importance of selecting the programming language and libraries when programming in parallel.

The rest of the paper is organized as follows: in Section 2 we present basic definitions about parallel programming. In Section 3 we analyze works where parallel programming languages have been compared. In Section 4 we present the algorithms used for the comparison and discuss aspects of their implementation. In Section 5 we present experiments and discuss the results. In Section 6 we conclude the paper and propose future work.

II. Definitions

A *program* is a sequence of instructions executed by a processor to do a specific task, e.g., calculate the sum of the elements of an array.

Manuscript received August 2, 2022; revised February 26, 2023.

Francisco Javier Moreno Arboleda is an associate professor at the Departamento de Ciencias de la Computación y de la Decisión, Universidad Nacional de Colombia, Sede Medellín, Colombia (phone: 604-425-5376; e-mail: fjmoreno@unal.edu.co).

Mateo Rincón Arias is a computer science engineer from Departamento de Ciencias de la Computación y de la Decisión, Universidad Nacional de Colombia, Sede Medellín, Antioquia, Colombia (e-mail: marinconar@unal.edu.co).

Jesús Antonio Hernández Riveros is an associate professor at Departamento de Energía Eléctrica y Automática, Universidad Nacional de Colombia, Sede Medellín, Antioquia, Colombia (e-mail: jahernan@unal.edu.co).

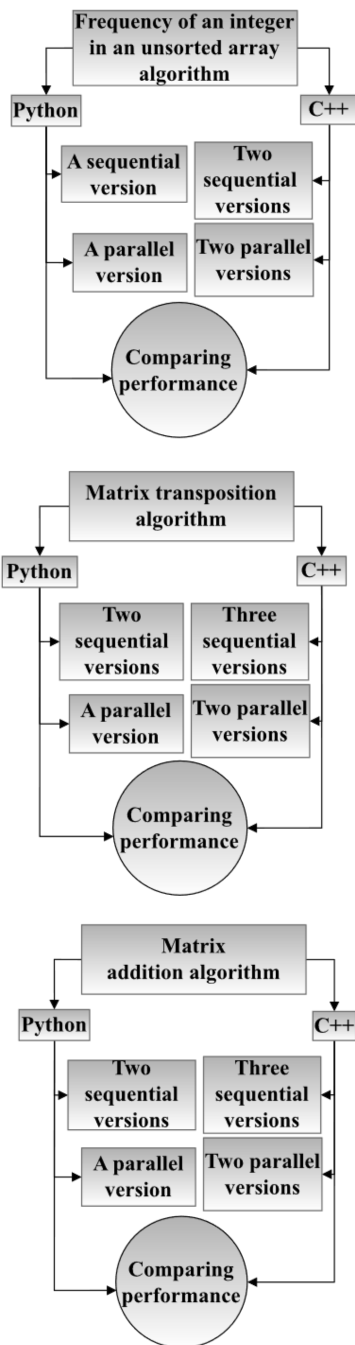

Fig. 1. Summary of our algorithm implementations

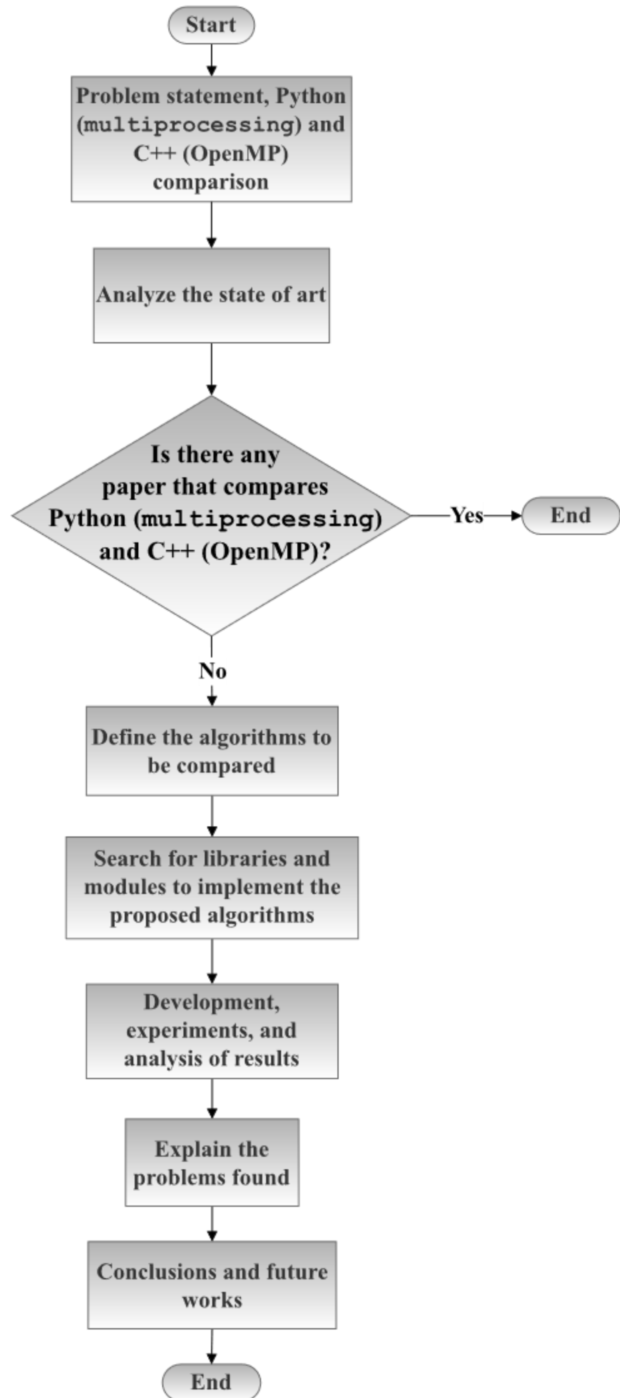

Fig. 2. Flowchart of the activities for the development of our work

A *process* is a program in execution. The execution time of a program is the total time required to execute it. A processor can execute only one instruction at a time.

The CPU (Central Processing Unit) is a processor and can have one or more cores. Each core is in turn a processor. A *thread* is a subsequence of instructions of a program executed at a specific time by a processor. For example, consider a loop that iterates over the 100 positions of an array (positions 0 to 99). One thread can handle iterations from positions 0 to 59 and another thread handles iterations from positions 60 to 99.

Although a processor can only be executing a program at a time, with threads it is possible to simulate the execution of several programs at a time. To do this, a program is divided into threads so that when a processor alternates between the threads of several programs, it seems that they are executed at a time. This exercise is called *multithreading* [18].

On the other hand, the GPU (Graphics Processing Unit) is also a processor that is responsible for rendering images on a computer, which lightens the load of the CPU. Just like a CPU, a GPU supports multithreading. Due to its

computation speed (in a computer the GPU is usually faster than the CPU, e.g., in a series of tests run by Buber [19], the Tesla k80 GPU was four to five times faster than the Intel Xeon Gold 6126 CPU), the GPU is often used to process general-purpose programs. This is called GPGPU (General Purpose Computing on GPU).

A *cluster* is a non-empty set of interconnected computers (nodes) that work as if they were a single system.

III. Related work

Hess et al. [20] compared OpenMP and MPI in a distributed shared memory cluster. They used 8 NEC 120Ed nodes, with dual processors composed by two 1Ghz Pentium III and 2 GB of memory in a Myrinet network 2000 NIC [21]. The comparison was made with three algorithms from the NAS Parallel Benchmarks (NPB) [22]: EP (Embarrassingly Parallel), CG (Conjugate Gradient), and FT (Fourier Transform). In general, OpenMP was twice as fast as MPI on all three algorithms.

McGinn and Shaw [23] used OpenMP and MPI to compare the performance of the Gaussian elimination algorithm (an algorithm for solving systems of linear equations). The algorithm was run on an IBM RS/6000 SP with four distributed nodes, each one with four processors. In OpenMP they used i) a shared memory environment (all threads have access to the same memory space, i.e., memory chunks) and ii) the *schedule* clause to specify the number of iterations that handle each thread in the loops. The number of variables in the system of equations was 400, 800, and 1200. In MPI, a distributed memory environment was used (each processor does not have access to the memory of the other processors; therefore, message passing was used). The results favored MPI, as it was 1.82 times faster than OpenMP.

Jost et al. [24] compared OpenMP and MPI in a SMP (Symmetric MultiProcessing) cluster. They used 16 Sun Fire 6800 nodes with 24 UltraSPARC-III Cu processors and 24 GB of shared memory on each node and four Sun Fire 15K nodes with 72 UltraSPARC-III Cu processors and 144 GB of shared memory on each node. The BT algorithm (Block Tri-diagonal solver) of the NPB was analyzed in MPI, in OpenMP, and in a hybrid way (i.e., using OpenMP and MPI in different proportions). In MPI, the comparison considered the interconnections between the nodes through Sun Fire Link (SFL) and Gigabit Ethernet (GE). Lower execution times with SFL were obtained. The hybrid implementation benefited more from the GE interconnections. In OpenMP the interconnections were not considered because OpenMP only supports shared memory. With both shared memory systems and interconnections, MPI had a better performance (although it is not clear how much), since it presented better scalability than the OpenMP and hybrid approaches.

Krawezik and Cappello [25] compared MPI and several programming styles in OpenMP (loop level, loop level with large parallel sections, and single program multiple data (SPMD)) for the NPB algorithms: CG, MG (Multi-Grid), FT, and LU (Lower-Upper Gauss-Seidel solver). They used an IBM SP3 Night Hawk II computer with 16 Power3+ 375 MHz and a SGI Origin 3800 computer with 128 R14000 500MHz multiprocessors. The results showed that OpenMP (in SPMD style) became 15% better than MPI in some algorithms and 140% better than the other OpenMP styles.

Drosinos and Koziris [26] proposed two MPI-OpenMP hybrid models to parallelize nested loops by means of *tiling* (a technique for partitioning a loop: *loop tiling* is also called *loop blocking* [27]) and then compared them with MPI. They used a Pentium III dual-SMP cluster of four nodes each with two Pentium III of 800 MHz and 256 KB of cache. The results did not show a clear winner.

Hochstein and Basili [28] compared OpenMP and MPI in an experiment with a group of eleven computer science students. Each student solved the sharks and fish problem [29] (a dynamics population problem containing a predator (shark) and a prey (fish) species) in OpenMP and MPI. Although each student had a computer, in the end the solutions were run on a dual-processor quad-core 2GHz Intel Xeon (but only one of the processors was used). The OpenMP solution represented an average time saving of 43% compared to MPI.

Wu and Taylor [30] compared the performance of MPI and the hybrid implementations (MPI and OpenMP) of the SP (Scalar Pentadiagonal solver) and BT algorithms of the NPB in large-scale multicore clusters: BlueGene/P, JaguarPF Cray XT5, and Jaguar XT4. The results showed that for the SP algorithm, the hybrid implementation performed up to 20% better than the MPI implementation; however, as the number of cores increased, MPI was 15% better. For the BT algorithm, the hybrid approach was 3% better and as the number of cores increased, MPI became 1% better.

Krpic, Martinovic, and Crnkovic [31] compared OpenMP and MPI using the naïve matrix multiplication (NMM) algorithm [32]. Matrices ranged from 2000 × 2000 to 5000 × 5000 with increments of 500. Three platforms were used for the experiments: a single core Intel P4 @ 2.4GHz, a dual core AMD Athlon @ 3.13GHz, and two Quad-core Intel Xeon E5430 @ 2.66 GHz. For the first computer the performance was similar. For the second computer OpenMP was better (using both one and two cores) for 3000 × 3000 matrices and of higher order and was even 50% faster than MPI. For the third computer, OpenMP was better for 2000 × 2000 matrices and of higher order becoming even twice as fast as MPI with 5000 × 5000 matrices.

Saravanan et al. [33] analyzed the performance of OpenMP in a multi-core CPU. They used a computer with an Intel Core i3 Dual core @ 2.13 GHz, 4 GB RAM, Windows 7, and Ubuntu 9.04. The NMM algorithm and an optimized matrix multiplication algorithm were evaluated (Strassen's Algorithm [34]), both in sequential and parallel implementations. Matrices ranged from 500 × 500 to 3000 × 3000 with increments of 500. For both algorithms, the execution time for the parallel implementation was lower. The NMM algorithm, in both implementations, was slower than Strassen's. For example, for matrices of 500 × 500 it was four times slower.

Reyes et al. [35] compared accULL [36], PGI Accelerator [37], and hiCUDA [38]. They evaluated four algorithms: i) lower-upper decomposition (an algorithm which factors a matrix as the product of a lower triangular matrix and an upper triangular matrix), ii) HotSpot (an algorithm to estimate the temperature of a processor), iii) Needleman-Wunsch (an algorithm to align DNA sequences), and iv)

NMM. They used a computer with an NVIDIA Tesla 2050 with 4GB memory and an Intel Core i7 930 of 2.80GHz. hiCUDA's performance was superior in all algorithms, even reaching 20 times better.

Lopes, Zeve, and dos Anjos [39] compared C, CUDA, OpenACC, and OpenMP. They used a computer running on Ubuntu 12.04, AMD Athlon(tm) II X2 270, and an NVIDIA GeForce GTX 650. They compared three algorithms: Mandelbrot set [40] (a fractal), N-Queens (a chess problem where N chess queens must be placed on an N×N chessboard so that no two queens attack each other), and NMM. The algorithms were implemented sequentially in C and in parallel in CUDA, OpenACC, and OpenMP. Each algorithm was executed ten times and the average execution time was obtained. For the NMM and Mandelbrot set algorithms, the execution time was similar in CUDA and OpenACC; in C it was 18 times slower, and in OpenMP it was 10 times slower. For the N-Queens algorithm, the execution time was similar in C, OpenACC, and OpenMP; and in CUDA was 32 times slower.

Xu, Chandrasekaran, and Chapman [41] used OpenMP to execute parallel programs on a CPU. They created threads, each one associated with a GPU which executes the code in OpenACC. For the experiments, they used an Intel Xeon and two Nvidia Tesla C2075. The algorithms evaluated were: S3D (a numerical simulation algorithm), NMM, and 2D heat conduction (an algorithm to calculate heat conduction in two dimensions). For the S3D and 2D heat conduction algorithms for grids with sizes of 4096, the performance with two GPUs was twice as good as with one GPU. For the NMM algorithm, the performance with two GPUs was slightly lower (it is not detailed by how much) than with one GPU for 1000 ×1000 matrices. However, for 5000 × 5000 matrices and of higher order, the performance with two GPUs was twice as good.

Kang et al. [42] compared OpenMP, MPI, and MapReduce on Hadoop. They evaluated two algorithms: all-pairs-shortest-path (find the shortest path between all pairs of vertices of a graph) and data-join (find Wikipedia pages in English associated with keywords written in English that appear in Wikipedia in Korean). For all-pairs-shortest-path they used graphs with 10, 100, and 1000 vertices. For data-join they used an XML file from Wikipedia in English with 4664819 articles of 4 GB and an XML file from Wikipedia in Korean with 529997 articles of 1.35 GB. They used five computers, each with an Intel Core i7-4770 3.40 GHz and 16GB RAM, with CentOS-6.4 Linux 64 bits. The execution time for all-pairs-shortest-path was twice as fast in OpenMP than in MPI on a single computer for all three cases and also for the case of 10 and 100 vertices for MPI in a cluster, made up of all five computers. With 1000 vertices, OpenMP was 36 times faster than MPI in the cluster. OpenMP was 1000 times faster than MapReduce in all cases. For data-join, MapReduce was 232 times faster than OpenMP and 337 times faster than MPI.

Kathavate and Srinath [43] analyzed the performance of parallelism in OpenMP. They implemented the NMM algorithm sequentially and in parallel. They used a computer with an Intel Pentium G630 dual-core and another with an Intel i7 dual-core. In both, each core can execute two threads. The parallelization with two threads showed an improvement of 46% compared to the sequential approach and the parallelization with four threads improved by 61%.

Holm, Brodtkorb, and Sætra [44] compared the performance, energetic efficiency, and usability of PyCUDA and PyOpenCL. They evaluated the algorithms for the shallow-water equations [45] with three numerical techniques (linear finite difference, nonlinear finite difference, and high-resolution finite volume). They used seven GPUs: Tesla M2090, K20, K80, P100, V100, GeForce GTX 780, and 840M. The two languages were similar in terms of performance and usability. Regarding energy efficiency, for the V100 and 840M GPUs, PyCUDA had a 30% better performance. For the other GPUs the power consumption was similar.

No works were found that compare the parallel performance of OpenMP and Python. This comparison is important because: a) Python is today one of the most used languages for data science applications, artificial intelligence, and machine learning [46] and b) few works analyze the performance of Python [47]–[49] and even fewer analyze its performance in terms of parallelism. For example, Wagner et al. [50] analyzed the parallel performance of Python considering the GPAW software (Grid-Based Projector Augmented Wave, a software to simulate electronic structures) and concluded that there is room for improvement with regard to C and Fortran.

For its part, OpenMP is an API for C/C++ and Fortran that allows algorithms to be parallelized using the CPU cores. This API also allows the programmer to parallelize regions (sections) of a program's code by means of preprocessing directives. The API has been developed since 1997 and is widely used in scientific computing [51]–[53]. Also, as of version 4.0, OpenMP supports GPGPU computing [54].

In Python, with the `multiprocessing` module we can generate subprocesses; which, unlike threads, use different *memory heaps* (the portion of memory available to a program) [55]. This module was added to the Python standard libraries in 2006 and was created to bypass the GIL (Global Interpreter Lock) [56], which is a form of mutual exclusion that prevents that multiple threads execute the same Python instructions at a time.

IV. Algorithms

For our experiments we evaluated three algorithms: i) frequency of an integer in an unsorted array, ii) matrix transposition (of a square matrix), and iii) matrix addition (of square matrices). Next, we present the pseudocodes of these algorithms.

Pseudocode of frequency of an integer in an unsorted array:

```
ALGORITHM
Frequency_integer_in_unsorted array(A, value)
Input:
A: unsorted array of size n.
value: integer to be searched in A.
Output:
frequency: frequency of value in A.
BEGIN
1. frequency = 0;
```

```
2. FOR i = 0 TO n - 1 DO [IN PARALLEL]
3.   IF Ai = value THEN
4.     frequency = frequency + 1;
5.   END IF
6. END FOR
END ALGORITHM
```

The complexity of this algorithm is O(n). The IN PARALLEL clause indicates that the instruction to which it is associated will be executed in parallel.

Pseudocode of matrix transposition:

```
ALGORITHM
Matrix_transposition(A)
Input:
A: n × n matrix.
Output:
AT: n × n matrix.
BEGIN
1. FOR i = 0 TO n - 1 DO [IN PARALLEL]
2.   FOR j = 0 TO n - 1 DO
3.     ATji = Aij;
4.   END FOR
5. END FOR
END ALGORITHM
```

The complexity of this algorithm is O(n^2).

Pseudocode of matrix addition:

```
ALGORITHM
Matrix_addition(A, B)
Input:
A: n × n matrix.
B: n × n matrix.
Output:
C: n × n matrix.
BEGIN
1. FOR i = 0 TO n - 1 DO [IN PARALLEL]
2.   FOR j = 0 TO n - 1 DO
3.     Cij = Aij + Bij;
4.   END FOR
5. END FOR
END ALGORITHM
```

Although the result of the matrix addition could be stored in one of the two input matrices (e.g., A = A + B), we used a third matrix C to work with independent memory regions. The complexity of this algorithm is also O(n^2).

V. Experiments

The algorithms were run on two computers, the first had an Intel (R) Core™ i7-9750H with 6 cores, an NVIDIA GeForce RTX 2060 @4.14 GHz, and 21 GB RAM; the second had an AMD Ryzen 5 3500U with 4 cores, a Radeon Vega Mobile Gfx @2.10 GHz, and 10 GB RAM.

We compared the execution time of the three algorithms as follows. First in Python, where we used these modules: `multiprocessing`, `functools` with its `partial()` function which facilitates the management of functions, `itertools` to iterate over data structures, e.g., arrays and matrices, `random` to generate random numbers, and `numpy` to operate on matrices.

TABLE I.
Data sample sizes

Frequency of an integer in an unsorted array			
35.000.000	50.000.000	57.500.000	65.000.000
Matrix transposition			
1125 × 1125	*2250 × 2250*	*3375 × 3375*	*4000 × 4000*
Matrix addition			
1063× 1063	2125 × *2125*	3188 × *3188*	4250 × *4250*

The algorithms were then run in C++ with OpenMP, `Vector` [57], and `Armadillo` [58]. `Vector` is a specialized sub-library for managing dynamic arrays and is part of the official C++ `Container` library [59]. `Armadillo` is a specialized library for linear algebra operations. Thus, we compared the results in C++ using different libraries.

The data to fill the arrays and matrices were random integers in the interval [0, 100]. The algorithms were evaluated with the values (size) shown in Table I.

Next, we describe our implementations.

In Python:

- **Frequency of an integer in an unsorted array:**
  A sequential implementation with `functools` and `random`.
  A parallel implementation with `multiprocessing`, `functools`, and `random`.
- **Matrix transposition:**
  Two sequential implementations: the first with `itertools` and `numpy` with explicit loops [60], i.e., the transposition was programmed using loops that iterate over the elements of the matrix and the second with `numpy`'s native function `transpose()`.
  A parallel implementation with `multiprocessing`, `itertools`, and `numpy`.
- **Matrix addition:**
  Two sequential implementations: the first with `itertools` and `numpy` (both with explicit loops) and the second with `numpy`'s native operator "+" to add objects of type `matrix`.
  A parallel implementation with `multiprocessing`, `itertools`, and `numpy` with explicit loops.

In C++:

- **Frequency of an integer in an unsorted array:**
  Two sequential implementations: the first with `Vector` and the second with `Armadillo`.
  Two parallel implementations: the first with `Vector` and OpenMP and the second with `Armadillo` and OpenMP.
  All four implementations with explicit loops.
- **Matrix transposition:**
  Three sequential implementations: the first with `Vector`, the second with `Armadillo` (both with explicit loops), and the third with `Armadillo`'s

native function `t()`, which computes the matrix transposition.
Two parallel implementations: the first with `Vector` and OpenMP and the second with `Armadillo` and OpenMP (both with explicit loops).

- **Matrix addition:**
Three sequential implementations: the first with `Vector`, the second with `Armadillo` (both with explicit loops), and the third with `Armadillo`'s native operator "+", which adds objects of type `matrix`.
Two parallel implementations: the first with `Vector` and OpenMP and the second with `Armadillo` and OpenMP (both with explicit loops).

Each of the implementations was executed ten times and the execution times were averaged, this average time was used for the analysis.

A. *Results of the first computer*

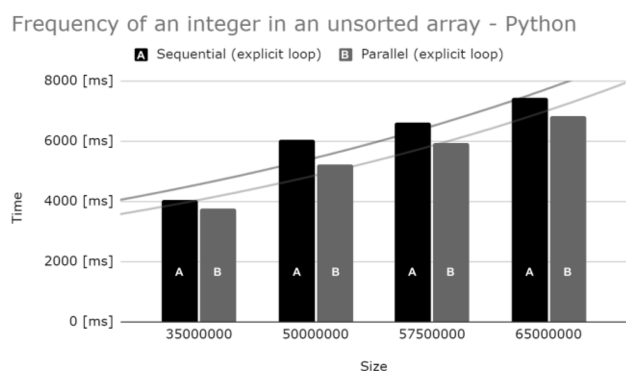

Fig. 3. Frequency of an integer in an unsorted array in Python (first computer)

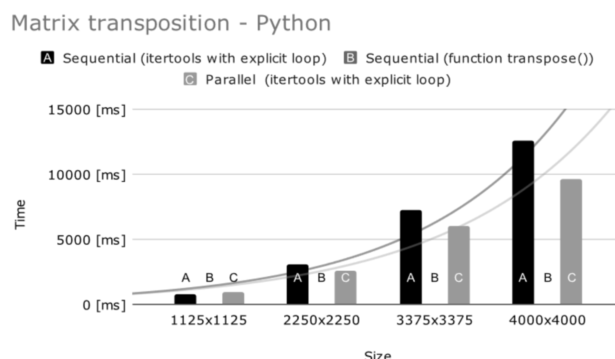

Fig. 4. Matrix transposition in Python (first computer)

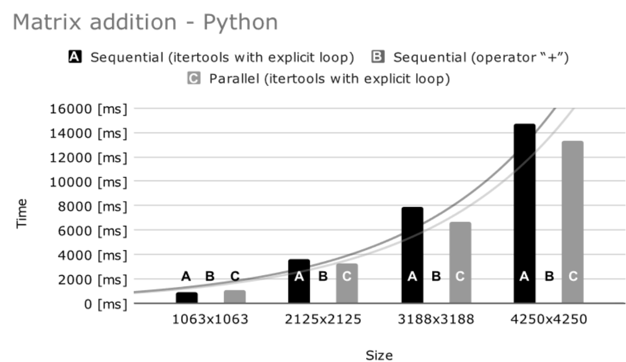

Fig. 5. Matrix addition in Python (first computer)

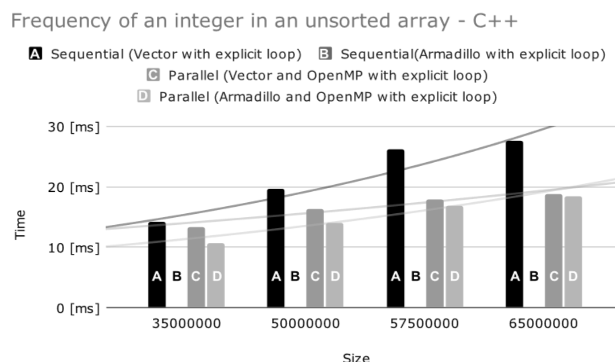

Fig. 6. Frequency of an integer in an unsorted array in C++ (first computer)

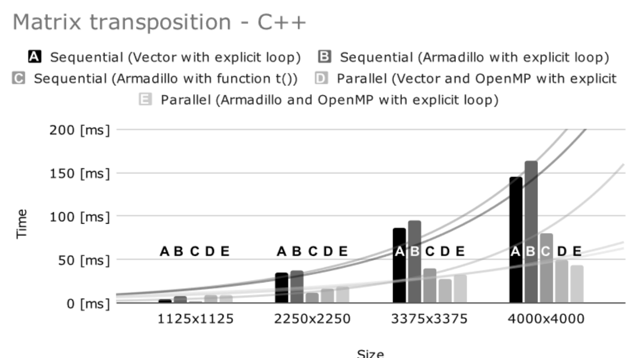

Fig. 7. Matrix transposition in C++ (first computer)

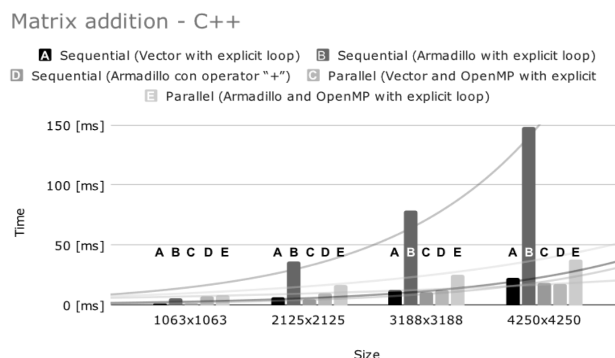

Fig. 8. Matrix addition in C++ (first computer)

Figures 3-8 correspond to the results of the first computer.

1) *Analysis of the results*

The sample sizes allowed us to observe the performance of the algorithms and their scalability according to the gradual increase in the size of the samples.

- **Frequency of an integer in an unsorted array**

The almost linear growth of all implementations stands out. That is, as the data increases, the slope of the curve grows smoothly (with a tendency to a line), unlike the other two algorithms (matrix transposition and matrix addition) where the trends suggest polynomial or exponentially increasing forms.

In Python the execution time was between 4 and 7 sec in its sequential implementation, when parallelizing we obtained an average improvement of 605 ms considering the four samples.

Regarding C++, in the sequential implementations, in `Armadillo` the execution time was 0 ms (see explanation

in subsection C), while in Vector it was 14 ms for the smallest sample and 28 ms for the largest. As for the two parallel implementations they had a similar performance: Armadillo was on average 1.56 ms better than Vector considering the four samples.

In conclusion, the parallel implementations reduced the times with respect to the sequential implementations (except the sequential implementation with Armadillo) by an average of 15% in both languages. C++ was up to 362 times better than Python.

- **Matrix transposition**

The sequential matrix transposition algorithm in Python using numpy's native function transpose() had an execution time of 0 ms. Except for this implementation, the parallel implementation using multiprocessing had the lowest time in Python (except for the smallest sample, possibly due to the parallelization costs for thread fork/join operations).

For its part, in C++ in the sequential implementations, Armadillo was the best thanks to its native function t(): it had execution times up to 64.39 ms lower than Vector. In the parallel implementations, Armadillo had the best execution times for the smallest and largest sample, while in the intermediate-sized samples Vector was on average 4.5 ms better than Armadillo.

In conclusion, the fastest implementation was Python's sequential using the native function transpose(). Except for this implementation, the parallel implementations in the two languages were better than the sequential ones, and C++ was up to 221 times better than Python.

- **Matrix addition**

The algorithm of the sequential matrix addition in Python using the native operator “+” had the lowest time. This is due to internal optimizations in the numpy library. On the other hand, the time of the parallel implementations with respect to the sequential ones improved as the size of the samples increased.

For its part, in C++ in the sequential implementations, Armadillo was the best thanks to the native operator “+”: it had execution times up to 4 ms lower than Vector. In the parallel implementations, Vector was on average 10 ms better than Armadillo. For the smaller samples, the parallel implementations had higher execution times than the sequential ones, possibly due to the parallelization costs for thread fork/join operations.

In conclusion, the fastest implementation was Python's sequential using the native “+” operator. Except for this implementation, parallel implementations in both languages were better than sequential ones for the largest samples, and C++ was up to 600 times better than Python.

2) *Conclusions regarding the first computer*

In general, the results showed that OpenMP was faster than Python in the parallel implementations. However, Python was the winner when using native functions and operators (transpose() and “+”) for matrix transposition and matrix addition.

B. *Results of the second computer*

Figures 9-14 correspond to the results of the second computer.

1) *Analysis of the results*

Again, the sample sizes allowed us to observe the performance of the algorithms and their scalability according to the gradual increase in the size of the samples.

- **Frequency of an integer in an unsorted array**

Similarly to the first computer, the almost linear growth of all implementations stands out.

In Python the execution time was between 3 and 6 sec for its sequential implementation, when parallelizing the execution times worsened on average by 4140.5 ms considering the four samples.

Regarding C++, in the sequential implementations, in Armadillo the execution time was 0 ms (just like in the first computer), while in Vector it was 10 ms for the smallest sample and 22 ms for the largest. As for the two parallel implementations, Vector was on average 10 ms faster than Armadillo considering the four sample sizes.

In conclusion, we observed a different behavior from the first computer. In Python, the parallel implementations increased the execution times with respect to the sequential ones by an average of 47.6%. This possibly occurred due to the decrease in the number of threads to parallelize operations, since the first computer has 12 threads and the second one 8.

- **Matrix transposition**

As in the first computer, the sequential implementation in Python using numpy's native transpose() function again had an execution time of 0 ms. In the other implementations, both the sequential and the parallel, Python had similar execution times (the differences were on average 250 ms, considering the four samples).

For its part, in C++ in the sequential implementations, Armadillo was the best; thanks to its native function t(): it had execution times up to 38 ms less than Vector. In the parallel implementations, Armadillo had the best execution times for the two smaller samples, while for the two larger samples, Vector was on average 20.8 ms better than Armadillo.

In conclusion, the fastest implementation was Python's sequential using its native function transpose(). Except for this implementation, the parallel implementations in both languages were better than the sequential ones (especially for larger arrays) and C++ was up to 380 times better than Python.

- **Matrix addition**

Similarly to the first computer, the sequential matrix addition algorithm in Python using the native operator “+” had the lowest execution time. On the other hand, the execution time of the parallel implementations with respect to the sequential ones improved as the size of the samples increased.

For its part, C++ in the sequential implementations, Armadillo was the best thanks to the operator “+”: it had execution times up to 16 ms better than Vector. In the parallel implementations, Vector was on average 753 ms

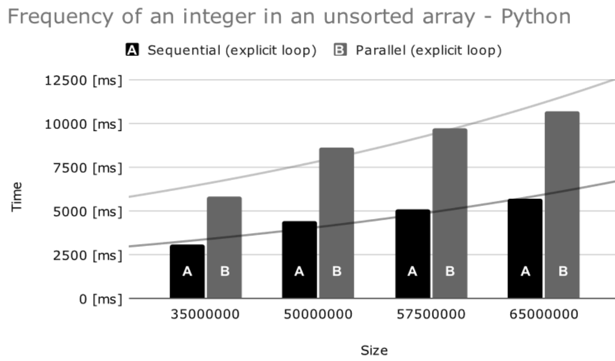

Fig. 9. Frequency of an integer in an unsorted array in Python (second computer)

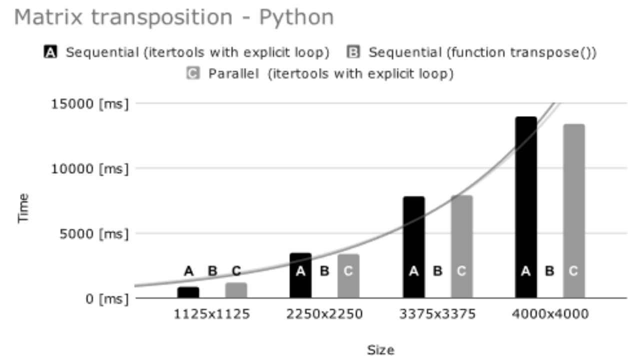

Fig. 10. Matrix transposition in Python (second computer)

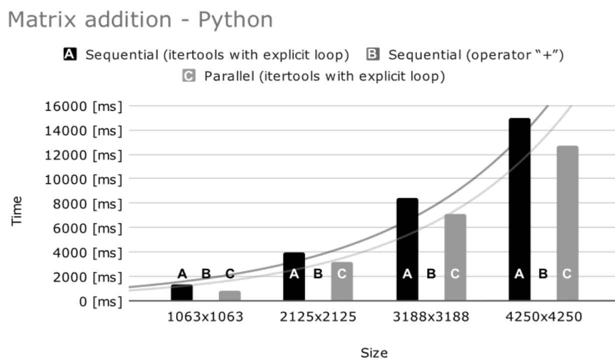

Fig. 11. Matrix addition in Python (second computer)

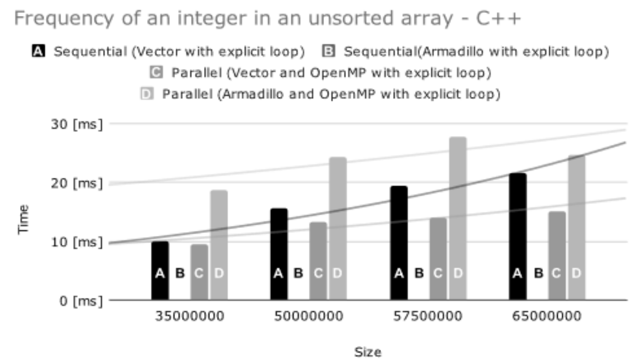

Fig. 12. Frequency of an integer in an unsorted array in C++ (second computer)

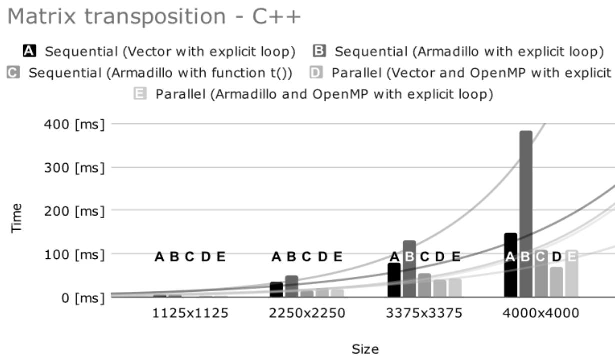

Fig. 13. Matrix transposition in C++ (second computer)

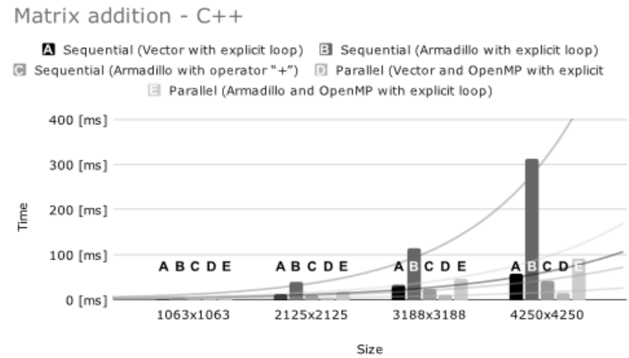

Fig. 14. Matrix addition in C++ (second computer)

better than `Armadillo`. Unlike the first computer, all the parallel implementations were better than the sequential ones.

In conclusion, the fastest implementation was the parallel one in C++ using OpenMP and `Vector`. Parallel implementations were found to outperform sequential ones for larger samples. Furthermore, unlike the other two algorithms, C++ and Python obtained similar results in their best implementation, differing on average by 0.51 ms.

2) Conclusion regarding the second computer

In general, the results are consistent with those of the first computer, except as indicated.

C. Problems

When conducting our experiments, we faced some problems. Mainly, these were related to sizes, memory space, parallelization, and the native functions or operators of the libraries.

The problems in Python mainly occurred in the matrix algorithms:

- **Matrix transposition:**
  `numpy`'s `transpose()` function cannot be parallelized as it is an internal function of the library. Indeed, when trying to parallelize it using the `map()` method of the `Pool` class of `multiprocessing`, which acts similarly to OpenMP's IN PARALLEL clause, we obtained incorrect results. In addition, in its sequential implementation `transpose()` already had execution times of 0 ms because this function *does not* physically transpose the data.
- **Matrix addition:**
  The `numpy` operator "+" cannot be parallelized since it is an internal operator of the library. Indeed,

when trying to parallelize it using the `map()` method of the `Pool` class of `multiprocessing`, we obtained incorrect results.

For its part in C++:

- **Frequency of an integer in an unsorted array:** The sequential implementation (`Armadillo` with explicit loop) had an abnormal behavior: execution times of 0 ms (the differences were perceived only at the level of microseconds). This behavior may be due to the efficiency of the library in traversing and indexing data structures.
- **Matrix transposition:** The `t()` function cannot be parallelized as it is an internal function of `Armadillo`. Indeed, when trying to parallelize it using the IN PARALLEL clause, we obtained incorrect results.
- **Matrix addition:** The operator "+" cannot be parallelized since it is an internal operator of `Armadillo`. Indeed, when trying to parallelize it using the IN PARALLEL clause, we obtained incorrect results.

In addition, when trying to work with larger data structures, Python programs generated errors due to lack of memory (RAM). This suggests that C++ manages memory better since it supported larger data samples. Table II shows the maximum limits found for the algorithms in C++. The limits found for the algorithms in Python were those used as the largest size in the experiments (see Table II). The complete results of our experiments can be seen in Tables AI and AII in the appendix. We also show in tables III and IV some additional experiments with the matrix multiplication algorithm, that confirm our previous conclusions about the performance of both languages and of both computers.

TABLE II
MAXIMUM DATA SAMPLE SIZES IN C++

Sequential	Parallel
Frequency of an integer in an unsorted array	
100000000	90000000
Matrix transposition	
10000 × 10000	10000 × 10000
Matrix addition	
10000 × 10000	10000 × 10000

VI. CONCLUSIONS AND FUTURE WORK

Computer 1 had better results on average for all algorithms. The results of both computers were consistent considering their differences in terms of resources.

From the previous analysis, the importance of selecting specialized modules, libraries, and APIs for certain algorithms is clear since these aspects decisively influence the execution times of the algorithms because they used to include native functions or operators that are more efficient than explicit implementations.

In summary, despite the different implementations and except for the cases where the native functions or operators had a better performance, we observed that C++ with OpenMP outperformed Python with its `multiprocessing` module.

In the future, we intend to create a visual tool along with a chatbot [61] in which analysts can enter an algorithm written in both languages, its parameters, and a data sample. Then, the tool will show execution statistics, and if possible, it will generate larger random data samples from the initial one and present a visual comparison of their performance results. We also plan to evaluate both languages with optimization algorithms for customers load schedule [62].

TABLE III.
RESULTS MATRIX MULTIPLICATION (FIRST COMPUTER)

Language	Implementation	Time (ms)		
	Size:	1000 × 1000	2000 × 2000	4000 × 4000
Python	Sequential (`itertools` with explicit loops)	1200	5200	21000
	Sequential (operator `matmul`)	0	0	0
	Parallel (`itertools` with explicit loops)	1350	4900	17500
C++	Sequential (`itertools` with explicit loops)	5	8	32
	Sequential (operator `matmul`)	4	9	27
	Parallel (`itertools` with explicit loops)	11	14	26

TABLE IV.
RESULTS MATRIX MULTIPLICATION (SECOND COMPUTER)

Language	Implementation	Time (ms)		
	Size:	1000 × 1000	2000 × 2000	4000 × 4000
Python	Sequential (`itertools` with explicit loops)	1800	7500	32000
	Sequential (operator `matmul`)	0	0	0
	Parallel (`itertools` with explicit loops)	2000	7200	25500
C++	Sequential (`itertools` with explicit loops)	7	11	39
	Sequential (operator `matmul`)	5	12	34
	Parallel (`itertools` with explicit loops)	16	19	33

APPENDIX

TABLE AI.
COMPLETE RESULTS (FIRST COMPUTER)

Language	Algorithm	Implementation	Time (ms)			
Python	Frequency of an integer in an unsorted array	**Size**	**35.000.000**	**50.000.000**	**57.500.000**	**65.000.000**
		Sequential (explicit loop)	4065	6058	6618	7446
		Parallel (explicit loop)	3769	5221	5927	6850
	Matrix transposition	**Size**	**1125 × 1125**	**2250 × 2250**	**3375 × 3375**	**4000 × 4000**
		Sequential (`itertools` with explicit loops)	808	3085	7233	12569
		Sequential (function `transpose()`)	0	0	0	0
		Parallel (`itertools` with explicit loops)	935	2545	5976	9603
	Matrix addition	**Size**	**1063 × 1063**	**2125 × 2125**	**3188 × 3188**	**4250 × 4250**
		Sequential (`itertools` with explicit loops)	903	3656	7864	14748
		Sequential (operator "+")	1	3	7	12
		Parallel (`itertools` with explicit loops)	1112	3262	6689	13287
C++	Frequency of an integer in an unsorted array	**Size**	**35000000**	**50000000**	**57500000**	**65000000**
		Sequential (`Vector` with explicit loop)	14	19	26	27
		Sequential (`Armadillo` with explicit loop)	0	0	0	0
		Parallel (`Vector` and OpenMP with explicit loop)	13	16	17	18
		Parallel (`Armadillo` and OpenMP with explicit loop)	10	14	16	18
	Matrix transposition	**Size**	**1125 × 1125**	**2250 × 2250**	**3375 × 3375**	**4000 × 4000**
		Sequential (`Vector` with explicit loops)	4	34	86	145
		Sequential (`Armadillo` with explicit loops)	7	37	94	163
		Sequential (`Armadillo` with function `t()`)	2	11	40	80
		Parallel (`Vector` and OpenMP with explicit loops)	9	17	28	49
		Parallel (`Armadillo` and OpenMP with explicit loops)	9	19	33	43
	Matrix addition	**Size**	**1063 × 1063**	**2125 × 2125**	**3188 × 3188**	**4250 × 4250**
		Sequential (`Vector` with explicit loops)	2	6	13	22
		Sequential (`Armadillo` with explicit loops)	6	37	79	148
		Sequential (`Armadillo` with operator "+")	1	4	11	18
		Parallel (`Vector` and OpenMP with explicit loops)	7	10	12	17
		Parallel (`Armadillo` and OpenMP with explicit loops)	9	17	25	38

TABLE AII.
COMPLETE RESULTS (SECOND COMPUTER)

Language	Algorithm	Implementation	Time (ms)			
Python	Frequency of an integer in an unsorted array	**Size**	**35000000**	**50000000**	**57500000**	**65000000**
		Sequential (explicit loop)	3070	4413	5068	5708
		Parallel (explicit loop)	5819	8644	9688	10670
	Matrix transposition	**Size**	**1125 × 1125**	**2250 × 2250**	**3375 × 3375**	**4000 × 4000**
		Sequential (`itertools` with explicit loops)	877	3470	7842	13918
		Sequential (function `transpose()`)	0	0	0	0
		Parallel (`itertools` with explicit loops)	1205	3388	7915	13371
	Matrix addition	**Size**	**1063 × 1063**	**2125 × 2125**	**3188 × 3188**	**4250 × 4250**
		Sequential (`itertools` with explicit loops)	1358	3927	8415	15013
		Sequential (operator "+")	2	3	10	16
		Parallel (`itertools` with explicit loops)	810	3217	7116	12681
C++	Frequency of an integer in an unsorted array	**Size**	**35.000.000**	**50.000.000**	**57.500.000**	**65.000.000**
		Sequential (`Vector` with explicit loop)	10	16	19	22
		Sequential(`Armadillo` with explicit loop)	0	0	0	0
		Parallel (`Vector` and OpenMP with explicit loop)	9	13	14	15
		Parallel (`Armadillo` and OpenMP with explicit loop)	19	24	28	25
	Matrix transposition	**Size**	**1125 × 1125**	**2250 × 2250**	**3375 × 3375**	**4000 × 4000**
		Sequential (`Vector` with explicit loops)	7	35	78	148
		Sequential (`Armadillo` with explicit loops)	5	50	130	383
		Sequential (`Armadillo` with function `t()`)	2	15	55	110
		Parallel (`Vector` and OpenMP with explicit loops)	4	20	40	71
		Parallel (`Armadillo` and OpenMP with explicit loops)	4	17	43	109
	Matrix addition	**Size**	**1063 × 1063**	**2125 × 2125**	**3188 × 3188**	**4250 × 4250**
		Sequential (`Vector` with explicit loops)	2	12	32	58
		Sequential (`Armadillo` with explicit loops)	4	39	115	313
		Sequential (`Armadillo` with operator "+")	2	11	24	42
		Parallel (`Vector` and OpenMP with explicit loops)	2	4	10	16
		Parallel (`Armadillo` and OpenMP with explicit loops)	3	17	45	91

REFERENCES

[1] M. E. Lesk. (1997). How much information is there in the world?" (Online). Available: https://www.lesk.com/mlesk/ksg97/ksg.html

[2] M. Amani et al., "Google Earth engine cloud computing platform for remote sensing big data applications: A comprehensive review," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 13, pp. 5326–5350, 2020.

[3] E. O'Neill. (2022). 10 companies that are using big data. *Institute of Chartered Accountants of Scotland*. (Online). Available: https://www.icas.com/news/10-companies-using-big-data

[4] A. Popovič, R. Hackney, R. Tassabehji, and M. Castelli, "The impact of big data analytics on firms' high value business performance," *Information Systems Frontiers*, vol. 20, no. 2, pp. 209–222, 2018.

[5] A. Woodie. (2015). Why big data is a 'how' at UPS, not a 'what'. *Datanami*. Available: https://www.datanami.com/2015/10/26/why-big-data-is-a-how-at-ups-not-a-what

[6] M. Samuels. (2017). Big data case study: How UPS is using analytics to improve performance, *ZDNET*. Available: https://www.zdnet.com/article/big-data-case-study-how-ups-is-using-analytics-to-improve-performance

[7] J. Dean and S. Ghemawat, "MapReduce: a flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[8] L. Clarke, I. Glendinning, and R. Hempel, "The MPI message passing interface standard," in *Programming Environments for Massively Parallel Distributed Systems*, K. Decker, Ed. Basel: Birkhäuser Basel, 1994, pp. 213–218.

[9] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science*

and Engineering, vol. 5, no. 1, pp. 46–55, 1998.
[10] OpenMP. (2022). OpenMP. *OpenMP*. (Online). Available: https://www.openmp.org
[11] F. Ciccozzi, L. Addazi, S. A. Asadollah, B. Lisper, A. N. Masud, and S. Mubeen, "A comprehensive exploration of languages for parallel computing," *ACM Computing Surveys*, vol. 55, no. 2, 2022.
[12] S. Bellamkonda, H.-G. Li, U. Jagtap, Y. Zhu, V. Liang, and T. Cruanes, "Adaptive and big data Scale parallel execution in Oracle," *VLDB Endowment*, vol. 6, no. 11, pp. 1102–1113, 2013.
[13] P. White. (2011). Understanding and using parallelism in SQL Server. *Redgate*, Available: https://www.red-gate.com/simple-talk/databases/sql-server/learn/understanding-and-using-parallelism-in-sql-server
[14] S. Shankar *et al.*, "Query optimization in Microsoft SQL Server PDW," *ACM SIGMOD International Conference on Management of Data*, Scottsdale, Arizona, 2012, pp. 767–776.
[15] C. K. Baru *et al.*, "DB2 parallel edition," *IBM Systems Journal*, vol. 34, no. 2, pp. 292–322, 1995.
[16] Python.org. (2022). Multiprocessing — process-based parallelism. *Python.org*. (Online). Available: https://docs.python.org/3/library/multiprocessing.html
[17] L. Prechelt, "An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl," *IEEE Computer*, vol. 33, no. 10, pp. 23–29, 2000.
[18] E. Rodríguez. (2017). Núcleos e hilos en un procesador: qué son y en qué se diferencian. *El Español*, (Online). Available: https://www.elespanol.com/omicrono/tecnologia/20170707/nucleos-hilos-procesador-diferencian/229478224_0.html
[19] E. Buber and B. Diri, "Performance analysis and CPU vs GPU comparison for deep learning," in *6th International Conference on Control Engineering & Information Technology*, Istambul, 2018, pp. 1–6.
[20] M. Hess, G. Jost, M. Müller, and R. Rühle, "Experiences using OpenMP based on compiler directed software DSM on a PC cluster," in *International Workshop on OpenMP Applications and Tools*, Toronto, 2003, pp. 211–226.
[21] N. J. Boden *et al.*, "Myrinet: a gigabit-per-second local area network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, 1995.
[22] NASA. (2022). NAS parallel benchmarks. *NASA Advanced Supercomputing Division*. (Online). Available: https://www.nas.nasa.gov/software/npb.html
[23] S. F. McGinn and R. E. Shaw, "Parallel Gaussian elimination using OpenMP and MPI," in *16th Annual International Symposium on High Performance Computing Systems and Applications*, Moncton, 2002, pp. 169–173.
[24] G. Jost, H.-Q. Jin, F. F. Hatay, and others, "Comparing the OpenMP, MPI, and hybrid programming paradigm on an SMP cluster," in *European Workshop on OpenMP and Applications*, Toronto, 2003, pp. 1–10.
[25] G. Krawezik and F. Cappello, "Performance comparison of MPI and OpenMP on shared memory multiprocessors," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 1, pp. 29–61, 2006.
[26] N. Drosinos and N. Koziris, "Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters," in *18th International Parallel and Distributed Processing Symposium*, Santa Fe, 2004, pp. 1–10.
[27] J. M. P. Cardoso, J. G. de Figueired Coutinho, and P. C. Diniz, *Embedded Computing for High Performance*. Morgan Kaufmann, New York, 2017, pp. 137–183.
[28] L. Hochstein and V. R. Basili, "A preliminary empirical study to compare MPI and OpenMP," ISI-TR-676, 2011.
[29] University of Cambridge. (2022). Population Dynamics - Part 5. *University of Cambridge - Faculty of Mathematics*. (Online). Available: https://nrich.maths.org/7274
[30] X. Wu and V. Taylor, "Performance Characteristics of Hybrid MPI/OpenMP Implementations of NAS Parallel Benchmarks SP and BT on Large-Scale Multicore Clusters," *The Computer Journal*, vol. 55, no. 2, pp. 154–167, 2012.
[31] Z. Krpic, G. Martinovic, and I. Crnkovic, "Green HPC: MPI vs. OpenMP on a shared memory system," in *35th International Convention MIPRO*, Opatija, 2012, pp. 246–250.
[32] S. Datta. (2021). Matrix multiplication algorithm time complexity. *Baeldung*. (Online). Available: https://www.baeldung.com/cs/matrix-multiplication-algorithms
[33] V. Saravanan, M. Radhakrishnan, A. S. Basavesh, and D. P. Kothari, "A comparative study on performance benefits of multi-core CPUs using OpenMP," *International Journal of Computer Science Issues*, vol. 9, no. 1, pp. 272–278, 2012.
[34] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull, "Strassen's algorithm for matrix multiplication: modeling, analysis, and implementation," *ACM/IEEE conference on Supercomputing*, Pittsburgh, 1996, pp. 6–9.
[35] R. Reyes, I. López, J. Fumero, and F. de Sande, "A comparative study of OpenACC implementations," *Jornadas Sarteco*, Elche, 2012, pp. 1–6.
[36] J. Lucas. (2013). Optimizing a stencil code with OpenACC. *WordPress*. (Online). Available: https://accull.wordpress.com
[37] PGroup. (2022). PGI compilers with OpenACC directives. *PGroup Compilers & Tools*. (Online). Available: https://www.pgroup.com/resources/accel.htm
[38] T. D. Han and T. S. Abdelrahman, "hiCUDA: high-level GPGPU programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 78–90, 2011.
[39] C. L. Ledur, C. M. Zeve, and J. C. dos Anjos, "Comparative analysis of OpenACC, OpenMP and CUDA using sequential and parallel algorithms," in *11th Workshop on Parallel and Distributed Processing*, Canoas, 2013, pp. 1–4.
[40] B. B. Mandelbrot, C. J. G. Evertsz, and M.C. Gutzwiller, *Fractals and Chaos: the Mandelbrot Set and Beyond*, Springer, New York, 2004.
[41] R. Xu, S. Chandrasekaran, and B. Chapman, "Exploring programming multi-GPUs using OpenMP and OpenACC-based hybrid model," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, Cambridge, 2013, pp. 1169–1176.
[42] S. J. Kang, S. Y. Lee, and K. M. Lee, "Performance comparison of OpenMP, MPI, and mapReduce in practical problems," *Advances in MultiMedia*, vol. 2015, pp. 1–9, 2015.
[43] S. Kathavate and N. K. Srinath, "Efficiency of parallel algorithms on multi core systems using OpenMP," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 3, no. 10, pp. 8237–8241, 2014.
[44] H. H. Holm, A. R. Brodtkorb, and M. L. Sætra, "GPU computing with Python: performance, energy efficiency and usability," *Computation*, vol. 8, no. 1, 2020.
[45] R. Sadourny, "The dynamics of finite-difference models of the shallow-water equations," *Journal of Atmospheric Sciences*, vol. 32, no. 4, pp. 680–689, 1975.
[46] M. D. Bloice and A. Holzinger, "A tutorial on machine learning and data science tools with Python," in *Machine Learning for Health Informatics: State-of-the-Art and Future Challenges*, A. Holzinger, Ed. New York: Springer International Publishing, 2016, pp. 435–480.
[47] G. Lanaro, *Python High Performance*. Packt Publishing, Birmingham, 2017.
[48] K. Lei, Y. Ma, and Z. Tan, "Performance comparison and evaluation of web development technologies in PHP, Python, and Node.js," in *2014 IEEE 17th International Conference on Computational Science and Engineering*, Chengdu, 2014, pp. 661–668.
[49] M. Gorelick and I. Ozsvald, *High Performance Python: Practical Performant Programming for Humans*. O'Reilly Media, Boston, 2020.
[50] M. Wagner, G. Llort, E. Mercadal, J. Giménez, and J. Labarta, "Performance analysis of parallel Python applications," *Procedia Computer Science*, vol. 108, pp. 2171–2179, 2017.
[51] D. Wang, C.-H. Wu, A. Ip, D. Wang, and Y. Yan, "Parallel multi-population particle swarm optimization algorithm for the uncapacitated facility location problem using OpenMP," in *IEEE Congress on Evolutionary Computation*, Hong Kong, 2008, pp. 1214–1218.
[52] S. Zhang, Z. Xia, R. Yuan, and X. Jiang, "Parallel computation of a dam-break flow model using OpenMP on a multi-core computer," *Journal of Hydrology*, vol. 512, pp. 126–133, 2014.
[53] H. Kasahara, M. Obata, and K. Ishizaka, "Automatic coarse grain task parallel processing on SMP using OpenMP," in *Languages and Compilers for Parallel Computing*, Berlin, 2001, pp. 189–207.
[54] K. Li. (2022). OpenMP accelerator support for GPUs - OpenMP. *OpenMP*. (Online). Available: https://www.openmp.org/updates/openmp-accelerator-support-gpus
[55] B. Noronha. (2017). Multithreading vs multiprocessing in Python. *DEV*. (Online). Available: https://dev.to/nbosco/multithreading-vs-multiprocessing-in-python--63j
[56] Python.org. (2023). Global Interpreter Lock. *Python.org*. (Online). Available:

https://wiki.python.org/moin/GlobalInterpreterLock
[57] cppreference. (2022). std::vector. *cppreference*. (Online). Available: https://en.cppreference.com/w/cpp/container/vector
[58] C. Sanderson and R. Curtin, “Armadillo: a template-based C++ library for linear algebra,” *Journal of Open Source Software*, vol. 1, no. 2, p. 1–7, 2016.
[59] Containers. (2022). *C++ Reference*. (Online). Available: https://www.cplusplus.com/reference/stl
[60] D. P. Clark. (2018). What’s the difference between implicit vs. explicit programming?. *CloudBess*. (Online). Available: https://www.cloudbees.com/blog/what-is-the-difference-between-implicit-vs-explicit-programming
[61] Chinedu Wilfred Okonkwo, and Abejide Ade-Ibijola, "Python-Bot: A Chatbot for Teaching Python Programming," Engineering Letters, vol. 29, no.1, pp25-34, 2021
[62] Sayawu Yakubu Diaba, Mohammed Elmusrati, and Miadreza Shafie-khah, "Evaluation of Optimization Algorithms for Customers Load Schedule," Lecture Notes in Engineering and Computer Science: Proceedings of The International MultiConference of Engineers and Computer Scientists 2021, 20-22 October, 2021, Hong Kong, pp122-127