ULPGC · UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

ParaBlock

Benchmarks of Parallel and Distributed Matrix Multiplication

Ricardo Juan Cárdenes Pérez[1]
ricardo.cardenes102@alu.ulpgc.es[1]

[1]University of Las Palmas de Gran Canaria
Faculty of Computer Sience
Campus de Tafira, 35017 Las Palmas
de Gran Canaria, Las Palmas

January 14^{th}, 2023

Abstract

This paper is aimed at finding a faster way to multiply large matrices, making use of the newest computer architectures. To do so, we introduce two different approaches: the tile matrix multiplication algorithm, which splits the work into different multiplication sub-processes, and a MapReduce version of Matrix Multiplication. The benefits of the tile multiplication algorithm underlie its ease for both vertical and horizontal scalability, while MapReduce is completely focused on horizontal scalability, but is easier to scale without refactoring a piece of code. In previous work, we achieved vertical scalability by implementing the sparse multiplication algorithm, but horizontal scalability was left behind. Now, to reach this purpose, we made use of both executor services, for parallelism, and Hadoop or even HazelCast for distributed programming, allowing us to deploy the matrix multiplication in a cluster with different machines. On the one hand, just applying parallelism to multiplication made us reach a noticeable out-perform in execution time when compared to the standard multiplication. More specifically, we obtained a speed-up of over 2.87x for a matrix of order 8192. MapReduce implementation, on the other hand, doesn't show any improvement due to the lack of sources for these experiments but would outperform tile multiplication in optimal conditions.

Keywords: *Matrix-Multiplication, Tile-Multiplication, Parallel-Computing, MapReduce*

1 Introducción

Matrix multiplication is a fundamental operation in numerous scientific and engineering applications, particularly in Big Data. With the burgeoning volumes of data being generated and analyzed across various fields, the efficiency of matrix multiplication holds paramount importance in accelerating computational tasks and optimizing data processing. This paper delves into the quest for enhancing the performance of matrix multiplication, especially when dealing with large ones. The exponential growth in data volumes poses a significant challenge for traditional matrix multiplication methods, urging the need for novel approaches that harness the capabilities of modern computer architectures.

The focus of this study lies in leveraging cutting-edge computing architectures to devise a faster and more scalable technique for multiplying large matrices. The introduction of the tile matrix multiplication algorithm stands as a pivotal aspect of this research. By decomposing the matrix multiplication task into distinct processes and aggregating the results, this algorithm seeks to deliver comparable outcomes to the standard matrix multiplication technique but with notable enhancements in computational efficiency.

In previous investigations, vertical scalability was achieved through the implementation of the sparse multiplication algorithm. However, achieving horizontal scalability remained an unaddressed challenge. In a concerted effort to overcome this limitation, this paper integrates executor services for parallelism and MapReduce for distributed programming. Throughout this article, we present empirical evidence showcasing the performance gains attained through parallelism and distributed computing techniques, culminating in a substantial speed-up for matrices of substantial orders.

2 Methodology

The experimental design of this study is centered around assessing the performance improvements achieved through the proposed tile matrix multiplication and the MapReduce multiplication algorithm when dealing with extensive matrices. The primary

goal is to leverage modern computing architectures to enhance the efficiency of matrix multiplication, specifically targeting large matrices, making use of the idle logical cores of CPU during the execution. First of all, it is necessary to have a clear idea of the algorithm that underlies these two matrix multiplications. So, let's take a look at them.

2.1 Tile Multiplication Algorithm

The fundamental concept underpinning our matrix multiplication strategy revolves around the notion of job splitting. Upon scrutinizing the conventional matrix multiplication paradigm, as elucidated in Benchmarks for Matrix Multiplication for Programming Languages [3], an intuitive approach emerges—partitioning the matrices to be multiplied into sub-blocks. This strategy allows us to perform the multiplication on these sub-blocks and achieve the same result as the standard algorithm.

In a mathematical context, consider two distinct matrices A and B, both of order $n \in \mathbb{N}$. We can partition these matrices into blocks of size $m \in \mathbb{N}$, where $n = k \cdot m$ for some $k \in \mathbb{N}$. Mathematically, this partitioning is represented as follows:

$$A \cdot B = \begin{pmatrix} A_{11} & \cdots & A_{1k} \\ \vdots & \ddots & \vdots \\ A_{k1} & \cdots & A_{kk} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & \cdots & B_{1k} \\ \vdots & \ddots & \vdots \\ B_{k1} & \cdots & B_{kk} \end{pmatrix} \quad (1)$$

Each entry in this resulting matrix structure is a dense matrix of size m, forming the basis for our multiplication approach. Algorithm 1 provides a detailed representation of this process.

The next crucial step in our approach involves the introduction of parallelism to further enhance computational efficiency. Parallelism is seamlessly applied by assigning each thread the computation of several rows of the matrices. This parallelization corresponds to the initial loop of the algorithm, where each thread handles a distinct subset of matrix rows concurrently.

In summary, our strategy capitalizes on job splitting through matrix partitioning, enabling parallelized computation of sub-blocks and significantly accelerating the matrix multiplication process, especially for large matrices. This approach leverages the capabilities of modern multi-core architectures, making it well-suited for the computational challenges posed by extensive datasets.

2.2 MapReduce Algorithm

The utilization of MapReduce in matrix multiplication holds significant importance for horizontally scaling systems, particularly when dealing with large datasets and computationally intensive tasks. Horizontal scalability refers to the ability to expand computational capacity by adding more machines or nodes to a distributed system, thereby enhancing overall performance and accommodating increased workloads.

In the context of matrix multiplication, the sheer size of matrices and the complexity of the computation demand efficient parallelization to achieve timely results. MapReduce provides an effective framework for horizontal scalability by distributing the matrix multiplication task across multiple nodes in a cluster. This approach divides the workload, enabling simultaneous processing of matrix elements on different machines.

Thus, the MapReduce Matrix Multiplication algorithm described here is designed to efficiently compute the product of two matrices in a distributed manner on a cluster of computers. It is implemented in Python with the MRJob framework and in Java with the Hadoop dependencies. In terms of mappers and reducers, the input of the program itself is expected in a specific format, where each line represents a matrix element with information about its type (A or B), row index, column index, and value. The algorithm begins with a Mapper function that parses the input and emits key-value pairs for each matrix element, with the key being a unique position identifier (i, k). The MapReduce framework then shuffles and sorts these pairs, ensuring that all values for a specific key are processed by the same Reducer.

The Reducer function receives the grouped pairs, separates values from matrices A and B, and performs the matrix multiplication by iterating over the columns and accumulating the product for each element in the resulting matrix. The final output consists of key-value pairs representing the matrix multiplication result, where the key denotes the position in the result matrix, and the value is the computed product. This algorithm leverages the distributed and parallel processing capabilities of MapReduce, making it suitable for addressing the computational challenges associated with large matrices in a scalable and efficient manner. Check algorithm 2 for more details.

Matrix multiplication with MapReduce is a fundamental technique employed, for example, in various machine learning frameworks to efficiently process large-scale computations. One specific example of matrix multiplication with MapReduce is in the computation of the gradient in distributed deep learning frameworks. During the training of neural networks, backpropagation involves matrix operations to calculate gradients for adjusting the model parameters. By distributing these computations across multiple nodes using MapReduce, the training process becomes significantly faster, allowing for the efficient exploration of large model architectures and datasets.

2.3 Implementation

In the course of our experimental investigation, we meticulously engineered the source code [1] not only to facilitate insightful comparisons with the standard matrix multiplication algorithm but also to embody the principles of SOLID design. This development pro-

cess included the creation of essential components such as the `DenseMatrix` and `BlockMatrix` classes. These classes, illustrated in Figure A1 in the appendix, play vital roles in executing dense matrix multiplication and tile matrix multiplication, respectively.

Expanding on prior work \cite{c2}, we designed the matrix builder's package with meticulous attention to detail, offering users a versatile toolset for constructing matrices from external sources. Each matrix in the model is paired with a dedicated builder, maintaining a consistent structure aligned with the conventions established in Benchmarks for Sparse Matrix Multiplication.

The operators module serves as the core of the application's functionality, adhering to SOLID principles. The module follows the Interface Segregation Principle (ISP) by designing precise and purposeful interfaces. Notably, transformers within this module facilitate seamless conversions between `BlockMatrix` and `DenseMatrix` objects, adhering to ISP's mandate of client-specific interfaces. Additionally, the Single Responsibility Principle (SRP) is conscientiously applied throughout the module. Multipliers, for instance, host the `multiply` method, accepting two matrices (Matrix interface instances) as inputs and algorithmically computing the result, adhering to SRP's focus on a single responsibility. Notably, tile multiplication is implemented in both the `BlockMatrixMultiplication` and `ParallelBlockMatrixMultiplication` classes, with the latter integrating parallelism to enhance computational efficiency. Refer to Figure A2 in the appendix for a comprehensive class diagram of this module, showcasing the application of SOLID design principles.

For the Java MapReduce implementation, we encapsulated the matrix multiplication process within the `MapReduceMatrixMultiplication` class. This class, depicted in the provided code snippet, handles the configuration, input/output paths, and execution of the MapReduce job. It employs the Hadoop framework to distribute the matrix multiplication task across a cluster. The `TextMapper` class manages the mapping phase, while the `TextReducer` class handles the reduction phase, performing the necessary computations for each matrix element. The implementation adheres to the principles of MapReduce, effectively harnessing parallel and distributed processing to horizontally scale the matrix multiplication operation. The integration allows for seamless execution on large datasets, making it well-suited for scenarios demanding horizontal scalability and efficient parallelization.

The Python implementation making use of MRJob framework made the job much easier, and can be seen on references, [5].

Algorithm 1: Tile Multiplication algorithm

Data: Two block splitted matrices A and B of size n and blocksize m
$C \leftarrow$ New block matrix of all zeros and size n
for *each* $i \in [1, n]_{\mathbb{N}}$ **do**
  for *each* $j \in [1, n]_{\mathbb{N}}$ **do**
    for *each* $k \in [1, n]_{\mathbb{N}}$ **do**
      $R \leftarrow A_{ik} \cdot B_{kj}$
      $c_{ij} \leftarrow c_{ij} + R$

Algorithm 2: MapReduce Matrix Multiplication

Data: Input matrices A and B, Matrix size $n \times n$
Mapper:;
for *each line in input* **do**
  Parse line to get matrix type, indices, and value;
  if *Matrix is A* **then**
    for *each k in [1, n]* **do**
      Emit key: (i, k), value: $('A', j, val)$;
  else
    for *each i in [1, n]* **do**
      Emit key: (i, k), value: $('B', j, val)$;
Reducer:;
for *each key, values* **do**
  Separate values into A and B matrices;
  Compute matrix multiplication for the key position;
  Emit key: position, value: result;

2.4 Code Decisions

In the course of code development, we made the deliberate choice to expunge generic classes from our implementation. While generic classes have proven advantageous in antecedent work [2], offering code conciseness and minimizing redundancy, nuanced considerations prompted our departure from this technology. Notably, our library's focal point on addressing Big Data challenges underscores the pivotal role of memory efficiency, as the matrices implicated in multiplication tasks often surpass heap and RAM limits.

The crux of the issue with generic classes lies in their necessity of Wrapper classes for data allocation, as opposed to the use of primitive data types. These Wrapper classes, in addition to housing the actual data, incur an overhead of 8 extra bytes (on 64-bit computer architectures) for the storage of the object pointer.

In the realm of Sparse Matrix Multiplication, this technical decision manifested in a substantial memory overhead. Allocating a sparse matrix of integers, for instance, resulted in a tripling of memory consumption in comparison to a scenario where generic classes were eschewed. This discrepancy arises from the inherent memory cost of Wrapper classes, with the 'int' primitive data type requiring only 32 bits for storage, contrasting with the 32+64 bits entailed by the Wrapper class Integer. In the case of 'double,' a similar doubling of required memory occurs when using the Double class.

As a result of these considerations, our decision to forego generic classes is rooted in the paramount need for memory optimization in the context of large-scale data processing. Consequently, the sparse matrix multiplication code is not included in this version of ParaBlock. However, it is slated for comprehensive refactoring and incorporation into subsequent versions, alongside the introduction of a parallelized version of sparse matrix multiplication.

2.5 Parallelism

As mentioned in section 2.1, parallelism could be obtained by splitting the first loop of Algorithm 1, so that each core can compute some of the rows separately and add their computations to the result. However, the real implementation parallels the computation of each i, j position in the resulting matrix, which allows us to achieve the same goal with an easier implementation. This way, made ourselves sure that no two logical cores share the same position, and by that avoided making use of semaphores, as the very nature of the code ensures mutual exclusion.

To implement such parallelism, we made use of executor services, which offer a robust and efficient mechanism for managing concurrent tasks. Executor services provide a higher-level abstraction over thread management and facilitate the parallel execution of tasks by handling the details of thread creation, pooling, and lifecycle management. By utilizing executor services, we enhance the scalability and maintainability of our parallelized tile matrix multiplication.

The dynamic allocation and reuse of threads provided by executor services contribute to minimizing overhead, optimizing resource utilization, and ultimately improving the overall performance of the parallel computation. Additionally, executor services offer flexibility in configuring the number of threads, allowing us to adapt the parallelization strategy to the available hardware resources and achieve optimal concurrency. This strategic use of executor services aligns with modern parallel computing best practices, fostering a streamlined and efficient parallel execution environment.

Now, for the experiment to run as fast as possible, we needed to make sure that the matrices to multiply were divided in such a way that the final number of blocks was a multiple of the number of logical cores available in the computer, and thus no processors are idle during execution. You can check this claim in Figure 4A in the appendix, which shows the use of different cores during the execution of the parallel multiplication. In contrast with dense matrix mult., Figure 3A, the process finished in the time frame captured in the image, and some time was even left, as can be seen.

3 Experiment

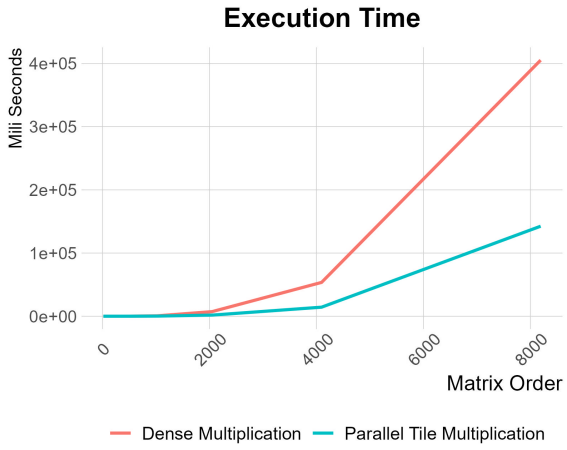
3.1 Hardware

For this task, we used a PC of brand MSI, a Katana model, with 16GB of RAM memory, solid state disk, and processor Intel Core i7 (16 cores). All windows will be closed while executing, except those needed to run the matrix multiplication code. The PC will be connected to power every time, as it is known that charging the PC turns to make the computer faster.

PC Model Katana GF66 11UE-486XES (MSI)
GPU NVIDIA® GeForce RTX™3050 Ti

3.2 Execution Time

To precisely measure the execution performance of the various implemented matrix multiplication operators, we employed the Java Microbenchmarking Harness (JMH). Within the benches packages in our test suite, we meticulously integrated benchmarking code for each of the matrix multipliers. Leveraging JMH allowed us to conduct rigorous and reliable performance evaluations, ensuring accurate measurements of the computational efficiency of our implementations.



(a) Dense and Parallel Tile Comp.

(b) Parallel Tile Transformations Comp.

Figure 1: Matrix Multiplication Time Comparisson

Order	MapReduce	MapReduce just mult.	Dense Mult.	Parallel Tile	Sparse Mu.
8	1775	1482	0.140	1.825	
16	1885	1474	1.17	1.566	0.21
32	2113	1452	9.351	4.520	1.581
64	5197	3440	70.22	27.459	2.526
128	20421	14528	641.399	219.452	10.551
256	120029	115087	7269.315	1775.87	43.54
512	1.32e+06	1.24e+06	53542.12	14325.23	248.23
1024			405318.29	142479.8	992.02

Table 1: Time execution in milliseconds for matrix multiplication in Java

As we can see in Figure 1a, by using tile matrix multiplication instead of standard multiplication and applying parallelism to it, we obtain a noticeable outperformance in terms of execution time. Even though it is not relevant for the lowest orders, **we reach the amount of** $2.84x$ **of speed-up** in favor of parallel tile multiplication **for matrix multiplication of order 8192**. One should also consider that the measured time not only includes the time it takes to multiply the matrices but also the time to make the transformations needed to perform such a multiplication. Figure 1b, in contrast, shows the execution time for parallel tile matrix multiplication when the input matrices are block matrices already, Par. Tile no transform., in contrast to the multiplication of matrices in dense matrix format using the same method. This doesn't have any real practical use. Usually, all matrices are given in dense format, so every time we want to make a multiplication we have to make that transformation. However, it is interesting to check how parallelism can help in performance, since the speed-up if we just focus on the multiplication process itself rises to $6.73x$, which was the parallelized part of the code. The overall observed acceleration aligns with Amdahl's Law, showcasing the significance of parallelization in enhancing overall performance, particularly in scenarios where the parallelizable fraction of the code constitutes a substantial portion of the computation.

Despite the apparent success in addressing large matrix multiplication challenges, a critical issue persists, notably concerning the storage and handling of sizable dense matrices in memory. The intricacies arise when dealing with matrices of substantial dimensions, where the memory requirements for storing these dense structures become a bottleneck.

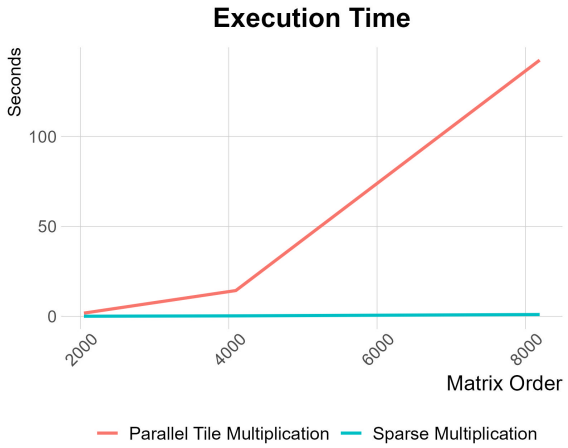


Figure 2: Parallel Tile and Sparse Comp.

The conventional approach to matrix multiplication often involves storing matrices entirely in memory. However, as matrices grow in size, the demand for memory space increases exponentially. Large dense matrices, with numerous elements, strain the available memory resources, potentially leading to performance degradation or even memory exhaustion in extreme cases.

The inherent limitations of memory-intensive operations become particularly pronounced when working with extensive datasets or in scenarios where real-time processing of large matrices is essential. In such cases, the conventional approach encounters constraints in terms of scalability and may lead to impractical memory usage.

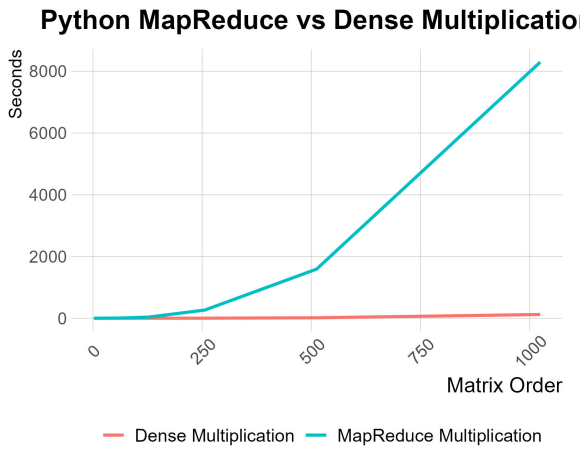
To overcome the challenges posed by the memory-intensive nature of tile matrix multiplication, we explored alternative solutions. Initially, Hazelcast emerged as a promising choice due to its in-memory data grid and distributed computing capabilities. The idea was to leverage Hazelcast to parallelize tile matrix multiplication tasks across a cluster of interconnected machines, thus harnessing the benefits of distributed computing.

However, a significant drawback surfaced during the exploration of Hazelcast. The necessity to store the entire matrix in the orchestrator, as outlined in the code [1], before initiating the tile matrix multiplication, posed a considerable limitation. This approach not only contradicted the fundamental purpose of distributing the computation but also intensified the memory constraints, especially when dealing with large matrices. Furthermore, as the experiment was conducted on a single computer, the anticipated improvements in multiplication speed were not realized, limiting the effectiveness of the Hazelcast implementation.

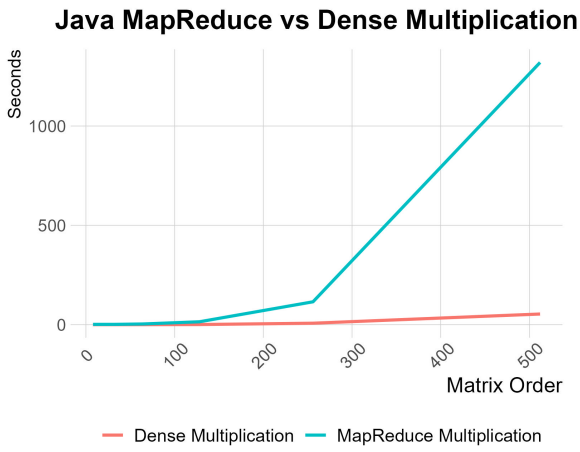
In light of these challenges, a more pragmatic solution became apparent: moving away from tile matrix multiplication altogether. Instead, we opted for a MapReduce approach, leveraging the capabilities of distributed computing offered by frameworks such as Hadoop. The MapReduce paradigm, with its ability to divide large computations into smaller, parallelizable tasks, provided a scalable solution without the need for storing the entire matrix in a single orchestrator.

The shift to MapReduce not only addressed the memory constraints associated with dense matrices, since allow us to define an input file where lines will be processed in different machines but also opened avenues for efficient parallelization across a distributed cluster of computers. This approach allowed us to break down the matrix multiplication task into manageable chunks, distributing them across multiple nodes, and most importantly, without needing to refactor a piece of code. The MapReduce implementation, while not demonstrating superior performance due to the lack of computers for these experiments, would achieve considerable performance improvements when scaled up to a cluster of Hadoop systems.

As one can appreciate in Figure 3a, and highlighted in table 2, Python's execution for MapReduce matrix multiplication drastically underperforms dense matrix multiplication and the same happens with Java, table 1 and Figure 3b. Even though the difference isn't that pronounced in Java, both languages seem to have a bad relation with MapReduce. However, this performance gap arises from several factors inherent to the MapReduce paradigm. The nature of distributed computing, coupled with the overhead involved in the Map and Reduce phases, introduces additional latency compared to the streamlined execution of dense matrix multiplication. Moreover, the need to serialize and deserialize data during the intermediate steps of MapReduce introduces computational overhead, impacting the overall efficiency.



(a) Python multiplication comparison

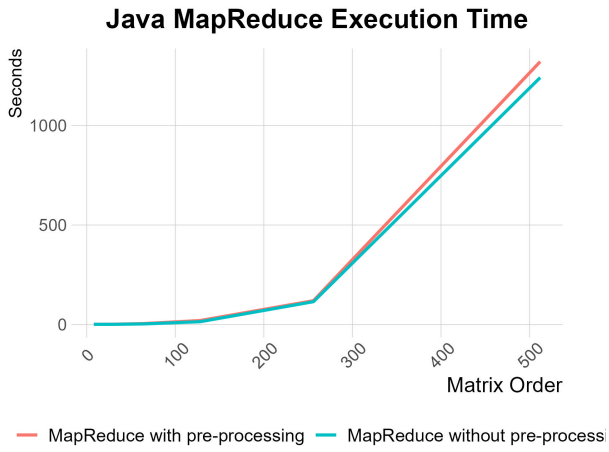


(b) Java multiplication comparison

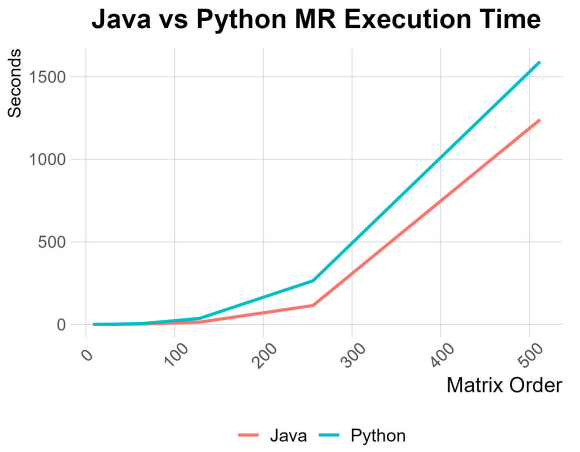
Figure 3: Dense and mapReduce multiplication comparison

Order	MapReduce Java	MapReduce Python
16	1474	1636.93
32	1452	2248.04
64	3440	6213.04
128	14528	36965.48
256	115087	264756
512	1.24e+06	1.59e+06
1024		

Table 2: Time execution languages comparison table in milliseconds for MapReduce multiplication



(a) Java MapReduce execution time



(b) Java vs Python MapReduce execution time

Figure 4: MapReduce execution times

The provided table details the execution times (in milli-seconds) for MapReduce and Dense matrix multiplication operations implemented in Python and Java across various matrix orders. Notably, the comparison between Python and Java reveals consistent trends. In the context of MapReduce operations, Python consistently exhibits longer execution times than Java across all matrix orders. The performance gap widens as the matrix order increases, emphasizing the relatively more efficient nature of MapReduce operations in Java. If compare both languages, attending to Table 2, **Java outperforms Python** with a **1.28x Speed-up**, which can be visually in Figure 4b. However, if we compare the results shown in table 1, Sparse Matrix Multiplication and Parallel Tile Multiplication suppose a Speed-up of over 5371x and 92.1x respectively for Java programming language.

4 Conclusions

In summary, this paper introduces parallel matrix multiplication algorithms as alternatives to outperform the traditional dense matrix multiplication, focusing on optimizing large matrix multiplication for contemporary computer architectures. The tile matrix multiplication algorithm, designed for both horizontal and vertical scalability, decomposes matrix multiplication into sub-block processes, enabling parallel computation. The primary result indicates a significant speed-up, surpassing 2.87x improvement in Java when compared to the standard algorithm, for a matrix of order 8192.

Moreover, the examination of CPU utilization, as depicted in Appendix Figure A4, reveals a crucial aspect of the algorithm's efficiency. The uniform distribution of CPU usage in Parallel Tile Matrix Multiplication, in contrast to the non-uniform distribution observed in dense matrix multiplication (Figure A3), highlights the algorithm's effective utilization of available computational resources.

However, when compared to the results achieved with the Sparse Matrix Multiplication algorithm, presented in previous work, and shown in detail in Table 4, the performance gap is apparent. Sparse matrix multiplication exhibits remarkable efficiency, outperforming the Parallel Tile Matrix Multiplication even in the parallelized setting developed. The current implementation, while achieving notable gains, falls short of matching the efficiency levels demonstrated by sparse matrix multiplication. Sparse matrices, with their inherent advantages in memory optimization and computational efficiency, continue to set a high standard for large-scale matrix operations.

Apart from this, tile matrix multiplication also faces inherent challenges related to memory management, especially when dealing with extensive matrices. As matrix sizes increase, the demand for memory space grows exponentially, potentially straining available resources and leading to runtime memory exceptions. The conventional approach of storing entire matrices in memory encounters limitations, particularly when handling large-scale data sets or real-time processing requirements. The intricacies of memory-intensive operations become pronounced, necessitating innovative solutions for efficient memory utilization.

In response to these memory challenges, our research delves into the transformative potential of the MapReduce paradigm. By breaking down matrix multiplication tasks into smaller, parallelizable chunks, MapReduce addresses memory constraints associated with dense matrices. This approach enables the distribution of computation across multiple nodes in a cluster, alleviating the burden on individual machines and contributing to efficient memory utilization, as the matrix doesn't need to be completely loaded in heap memory on a single computer. And even though we obtained a noticeable underperform concerning the other seen algorithms, the MapReduce framework, with its capacity for scalable and distributed computing, emerges as a promising solution to overcome the memory hurdles faced by traditional matrix multiplication methods.

Moreover, the exploration of MapReduce's applications extends beyond memory efficiency. The paradigm's ability to parallelize computations and operate on distributed data sets holds significant implications, particularly in the realm of big data and machine learning. Matrix multiplication, a fundamental operation in machine learning algorithms, stands to benefit from the scalability and parallelization offered by MapReduce. The efficient processing of large matrices through MapReduce could usher in a new era of advancements in data-driven applications, unlocking the potential for faster and more scalable computations in fields reliant on extensive matrix operations.

Future Work

Moving forward, our focus will shift towards parallelizing Sparse Matrix Multiplication, leveraging the algorithm's strengths in scenarios involving matrices with a substantial number of zero elements. The efficiency of Sparse Matrix Multiplication stems from its ability to skip computations involving zero values, leading to significant time complexity improvements, particularly when dealing with sparse matrices that predominantly consist of zeros. By introducing parallelization to this algorithm, we aim to enhance its performance across various matrix structures, including dense, tile, and sparse matrices with sparse patterns. Moreover, our plan involves the integration of Parallel Sparse Matrix Multiplication into a distributed computing environment using MapReduce. frameworks.

Frequent Item Set Algorithm

As another use of MapReduce in Data Science and Engineering, we introduce the Frequent Item Set algorithm, a fundamental technique in data mining de-

signed to uncover patterns of frequent co-occurrence within transactional datasets. Operating on a set D of transactions, the algorithm aims to identify sets F of items with support greater than a specified threshold s. Formally, F is considered frequent if $\text{support}(F) \geq s$, where the support of F represents the proportion of transactions in D containing F. The algorithm explores all possible combinations of elements in D, utilizing methods such as Apriori or FP-Growth to streamline the search process and ensure computational efficiency in discovering frequent patterns.

The utility of the Frequent Item Set algorithm lies in its ability to reveal meaningful associations among elements in extensive transactional datasets. For instance, in an e-commerce dataset, it can identify frequent itemsets, such as pairs or triples of products often purchased together. This information is valuable for marketing strategies, enabling personalized recommendations, and informing business decisions based on customer purchasing patterns.

The complexity of the Frequent Item Set algorithm depends on factors like dataset element number (N) and the specified support threshold (s). In the worst-case scenario, the exhaustive search of all possible combinations results in exponential complexity expressed as $O(2^N)$. However, optimization techniques, such as pruning infrequent sets, can significantly enhance efficiency, reducing the complexity to $O(m \cdot N)$, where m is the average number of elements in a frequent set. In Figure 5a, we can observe the correlation between the execution time of the Frequent Item Set algorithm and the growth in the number of transactions. As the dataset scales, the algorithm's performance is impacted, demonstrating the importance of considering computational efficiency for large-scale data processing. Moreover, Figure 5b shows how the execution time grows with the total number of elements, for a fixed number of 16 transactions. We also show the difference regarding different numbers of items per rule but had no test data for more elements than 32. We obtained these transaction examples used to test using ChatGPT, with prompts similar to: 'make 16 combinations of different sizes for a total of 25 fruits for the frequent item set algorithm'

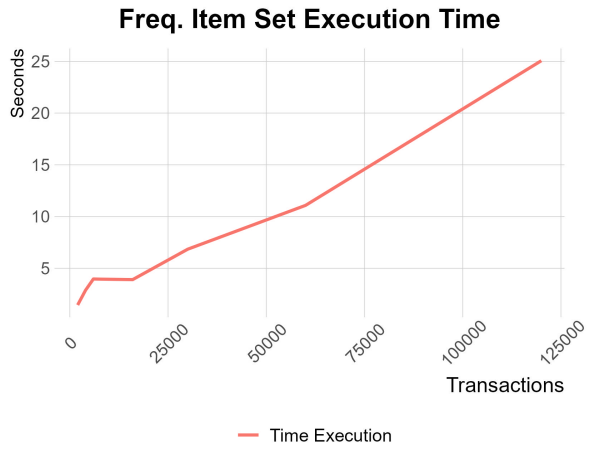
Turns out MapReduce can enhance the efficiency of the Frequent Item Set algorithm in handling large-scale datasets. By distributing the computation across multiple nodes, MapReduce enables parallel processing, reducing the overall execution time. For example, during the generation of candidate itemsets, MapReduce can efficiently distribute the workload, ensuring scalability and faster identification of frequent patterns.

The provided Java code, [4], consists of three main classes: FrequentItemSet, Mapper, and Reducer. The FrequentItemSet class configures and initiates a MapReduce job using Hadoop. It sets up the necessary parameters, such as the mapper and reducer classes, and specifies input and output paths.

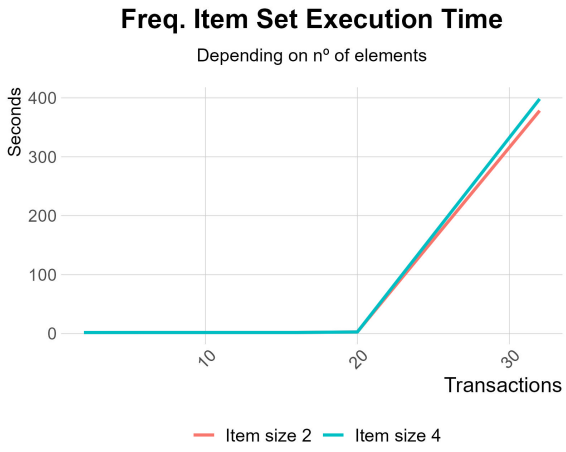
The Mapper class, an extension of the abstract class IntMapper, defines the logic for processing input transactions. It sorts and removes duplicates from the items in each transaction. The selectItems method is then invoked to identify frequent itemsets of the size given in the main class' parameter.

The Reducer class, an extension of the abstract class IntReducer, aggregates intermediate results. It sums the counts of occurrences for each frequent item set and emits the final output if the count exceeds a predefined threshold.

This modular design allows for efficient identification of frequent itemsets in large transactional datasets through the MapReduce paradigm, providing scalability and parallel processing capabilities. Also, the Python code, [5], implements such a code making use of the MRJob framework, which reduces the code to one single MRJob extension class.



(a) Ex. time depending on transactions



(b) Ex. time depending on elements

Figure 5: Frequent Item Set time complexity

References

[1] Cárdenes Pérez, Ricardo J. (2023) ParaBlock Source Code
https://github.com/ricardocardn/ParaBlock

[2] Cárdenes Pérez, Ricardo J. (2023) Benchmarks of Sparse Matrix Multiplication
https://github.com/ricardocardn/SparseMatrixMultiplication

[3] Cárdenes Pérez, Ricardo J. (2023) Benchmarks of Matrix Multiplication
for Different Programming Languages
https://github.com/ricardocardn/SparseMatrixMultiplication

[4] Cárdenes Pérez, Ricardo J. (2024) MapReduce implementation of Frequent Item Set algorithm
https://github.com/ricardocardn/MRFrequentIemSet

[5] Cárdenes Pérez, Ricardo J. (2024) Python MapReduce implementation of
Matrix Multiplication and Frequent Item Set algorithm
https://github.com/ricardocardn/MapReduceMatrixMult

APPENDIX

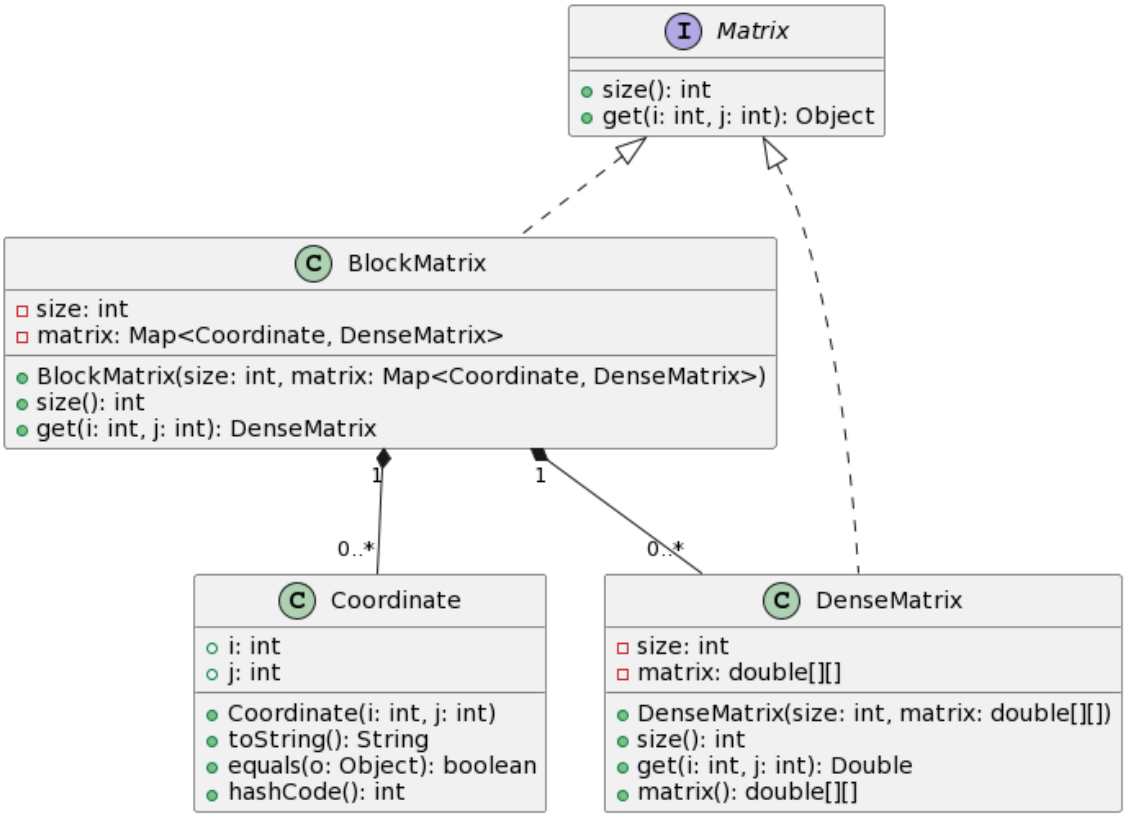


Figure A1: Model UML Diagram

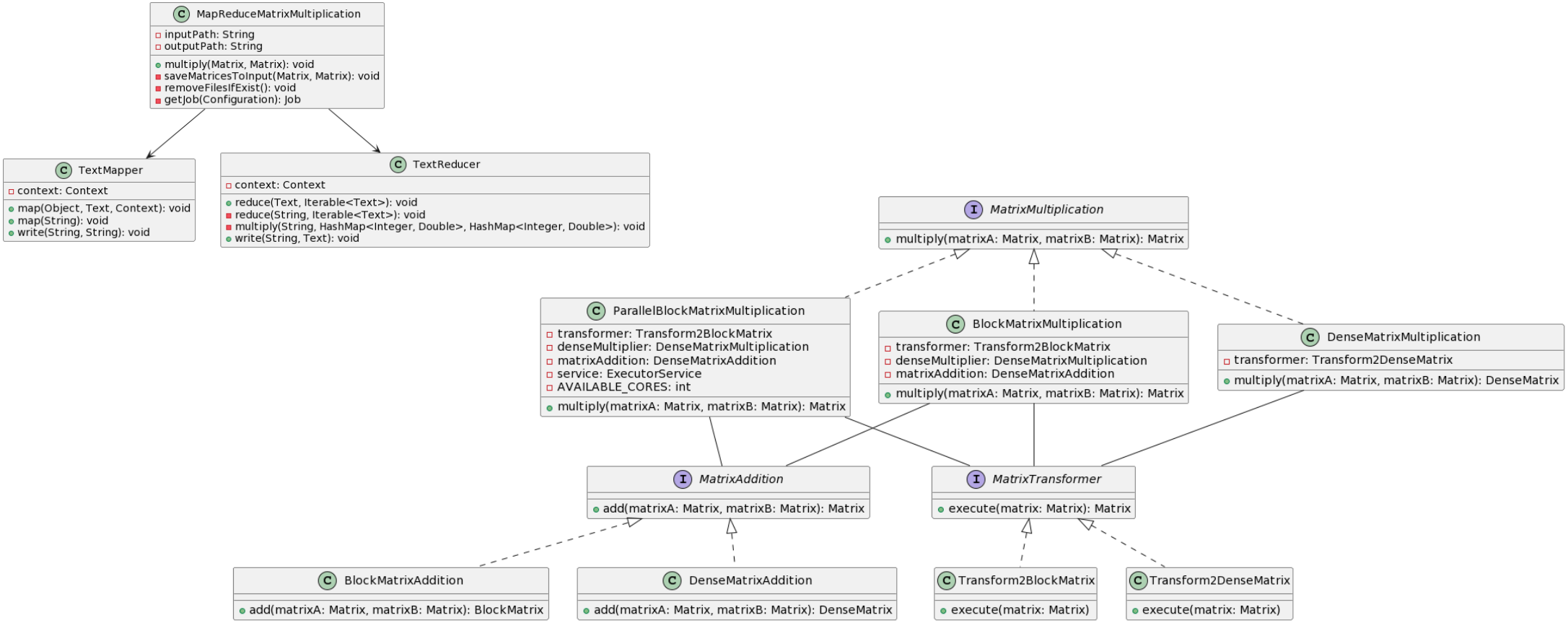


Figure A2: Operators UML Diagram

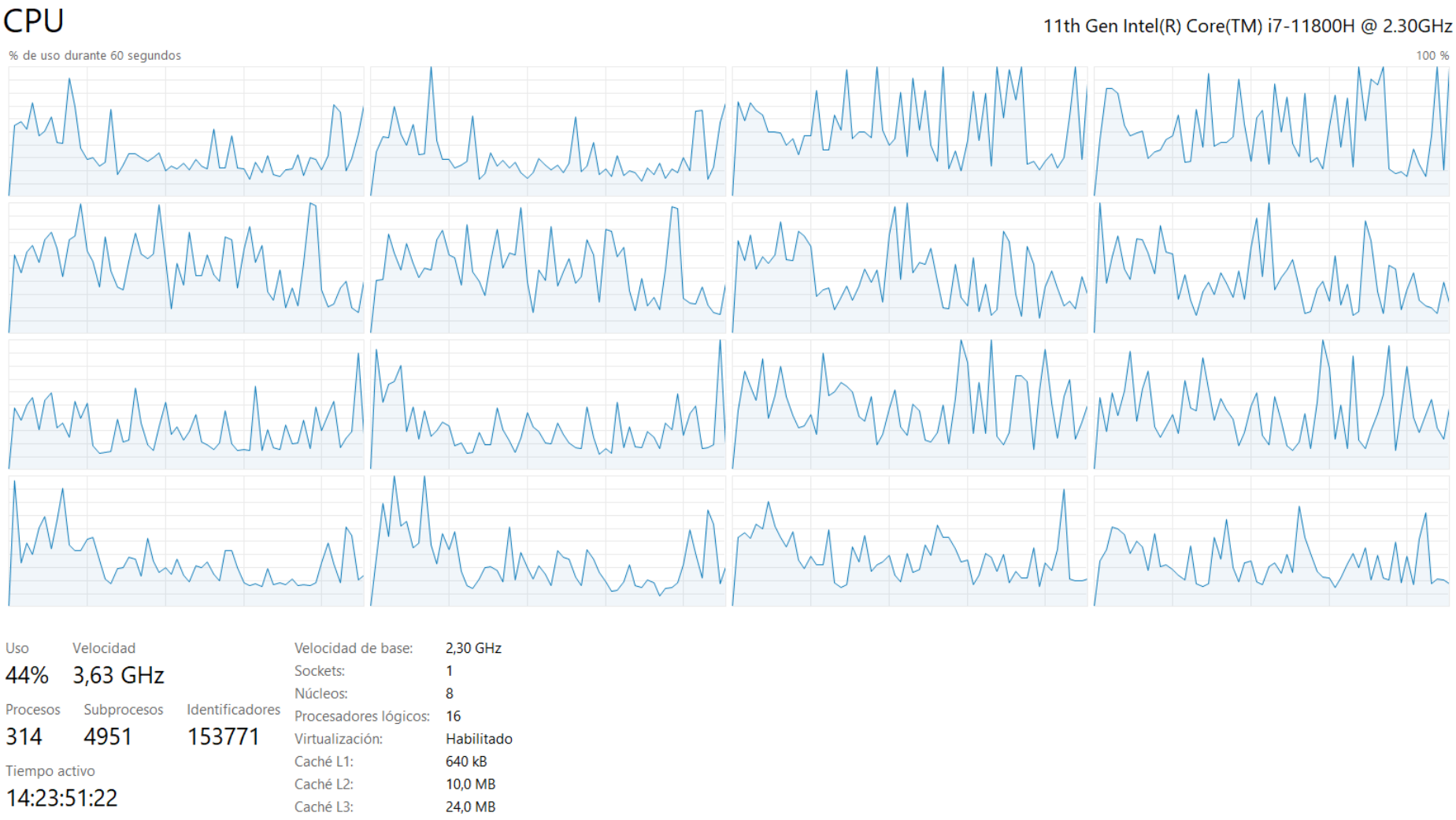


Figure A3: Dense Multiplication CPU Distribution

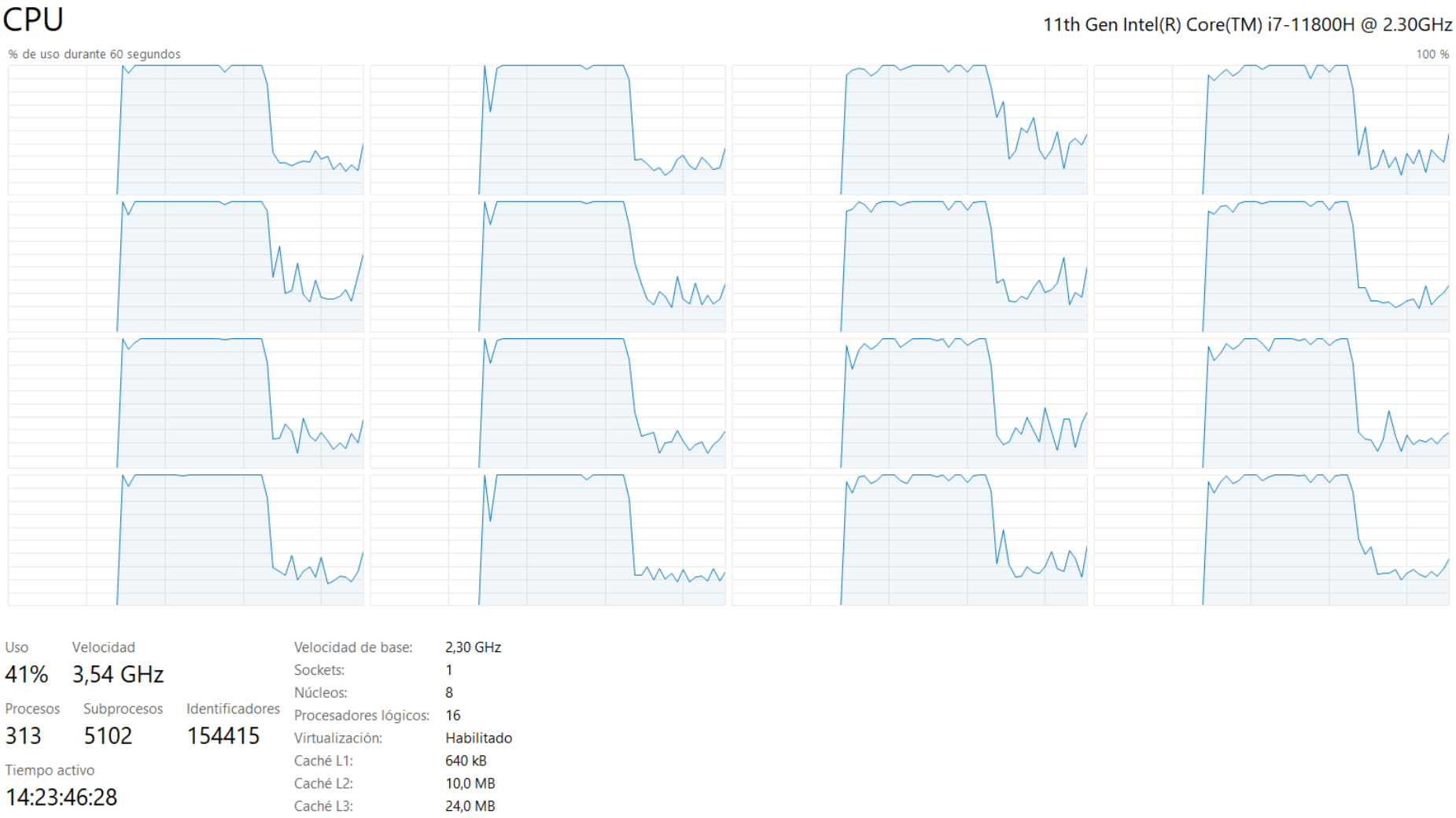


Figure A4: Parallel Multiplication CPU Distribution