Benchmarks of Parallel Tile Matrix Multiplication

Ricardo Juan Cárdenes Pérez[1]

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

December, 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 the tile matrix multiplication algorithm, which splits the work into different multiplication sub-processes, adding up all results when finished, and getting the same final result as the standard matrix multiplication. The benefits of this algorithm underlie its ease for both vertical and horizontal scalability. 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 executor services, for parallelism, and 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. HazelCast, on the other hand, doesn't show any improvement due to its latency time and the fact that memory is bounded for large matrices, where the effects would start to seem relevant.

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

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 HazelCast 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 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 this fast matrix multiplication. So, let's take a look at it.

2.1 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 Implementation

In the pursuit of this experimental investigation, we meticulously engineered the source code [1] to not only facilitate insightful comparisons with the standard matrix multiplication algorithm but also to embody the principles of SOLID design. The development encompassed the creation of essential components, namely the DenseMatrix and BlockMatrix classes, each playing a pivotal role in executing dense matrix multiplication and tile matrix multiplication, respectively. For a comprehensive understanding of these classes, consult Figure A1 in the appendix.

Building upon the foundation laid in previous work [2], we meticulously designed the matrix builder's package, providing users with a versatile toolset for constructing matrices from external sources. Each matrix in the model is paired with a dedicated builder, maintaining a consistent structure aligning with the conventions established in Benchmarks for Sparse Matrix Multiplication.

The operators module serves as the heartbeat of the application's functionality, adhering to SOLID principles. The application 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. Significantly, tile multiplication is implemented in both the BlockMatrixMultiplication and ParallelBlockMatrixMultiplication classes, with the latter integrating parallelism to enhance computational efficiency. Delve into Figure A2 in the appendix for a comprehensive class diagram of this module, showcasing the application of SOLID design principles.

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
            $R \leftarrow A_{ik} \cdot B_{kj}$
            $c_{ij} \leftarrow c_{ij} + R$
```

2.3 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.

2.4 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.

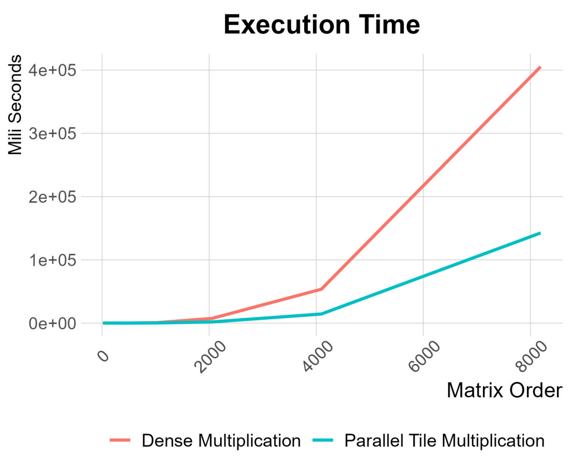
The dynamic allocation and reuse of threads provided by executor services contribute to minimizing overhead, optimizing resource utilization, and ultimately improving the overall performance of the parallel computation. Additionally, executor services offer flexibility in configuring the number of threads, 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.

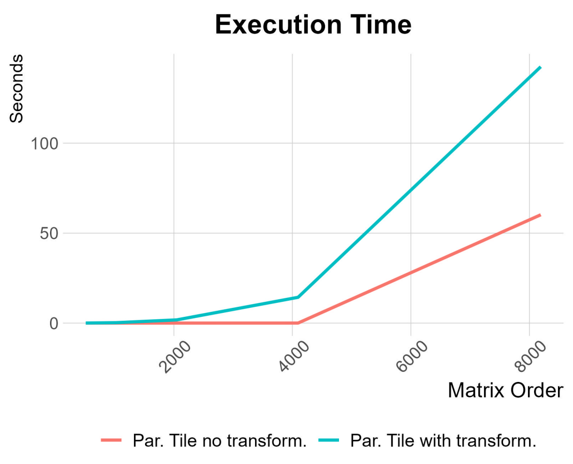
3 Experiment

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.

JMH provides a robust framework specifically designed for benchmarking Java code, offering features such as warm-up iterations, multiple iterations, and statistical analysis of results. By incorporating JMH into our testing infrastructure, we achieved a systematic and controlled environment for assessing the execution times of the matrix multiplication operations. The benchmarking code within the benches packages follows best practices for conducting fair and unbiased performance evaluations, taking into account factors like JVM warm-up effects and ensuring stable and reproducible results.



(a) Dense and Parallel Tile Comp.



(b) Parallel Tile Transformations Comp.

Figure 1: Matrix Multiplication Time Comparisson

Order	Dense Mult.	Parallel Tile	Sparse Mu.
16	0.003	0.324	x
32	0.022	0.583	x
64	0.140	1.825	x
128	1.17	1.566	0.21
256	9.351	4.520	1.581
512	70.22	27.459	2.526
1024	641.399	219.452	10.551
2048	7269.315	1775.87	43.54
4096	53542.12	14325.23	248.23
8192	405318.29	142479.8	992.02

Table 1: Time execution in milliseconds for matrix multiplication

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.

It seems like we have solved the problem of large matrices multiplication and in a pretty intuitive way, nothing further from the truth. When we compare with the results obtained for sparse matrix multiplication given in work [2], which measures are also shown in Table 1 and compared in Figure 2, we realize how far we are from that performance.

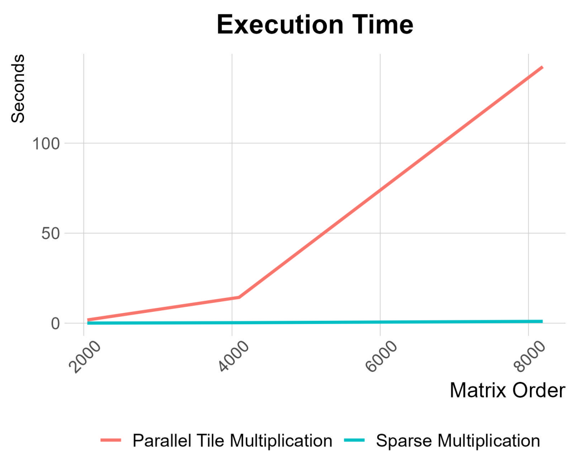


Figure 2: Parallel Tile and Sparse Comp.

This library offers a way to deploy the parallel tile multiplication in a cluster of computers, in pursuit of achieving the performance that sparse matrices gave us. However, we only had two computers for this experiment, and with two matrices of order 10000, the shared RAM reserved for HazelCast was already full.

Probably, **memory usage is the worst aspect of using dense matrices**. Java heap memory is not big enough to allocate the entire dense matrices larger than a certain order. Actually, in the device where this experiment took place, matrices larger than 18000 rows and columns can be no longer multiplied. Because of this reason, sparse matrix multiplication will be our main focus point later on, even though they are only efficient when the matrices are null for many entries.

4 Conclusions

In summary, this paper introduces the Parallel Tile Matrix Multiplication algorithm, focusing on optimizing large matrix multiplication for contemporary computer architectures. The 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 for a matrix of order 8192 when parallelized.

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, the performance gap is apparent. Sparse matrix multiplication exhibits remarkable efficiency, outperforming the Parallel Tile Matrix Multiplication even in a parallelized setting. 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.

This study highlights the challenges of achieving comparable performance with sparse matrix multiplication and emphasizes the need for further optimizations and exploration of advanced distributed computing infrastructures. While the Parallel Tile Matrix Multiplication shows promise, future work may delve deeper into strategies to narrow the performance gap and enhance its applicability in large-scale computing scenarios.

Future Work

Moving forward, our focus will shift towards parallelizing Sparse Matrix Multiplication, leveraging the advantages of the algorithm for both dense, tile and sparse matrices. The goal is to develop a parallelized solution that ensures optimal performance across various matrix structures. Additionally, we plan to integrate Parallel Sparse Matrix Multiplication into a distributed computing environment using HazelCast, as was done for this paper for tile multiplication. This step aims to address memory constraints by distributing matrix multiplication tasks across a cluster of computers, paving the way for scalability and efficient handling of larger matrices. These efforts align with the evolving landscape of computational challenges, providing valuable contributions to the field.

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

APPENDIX

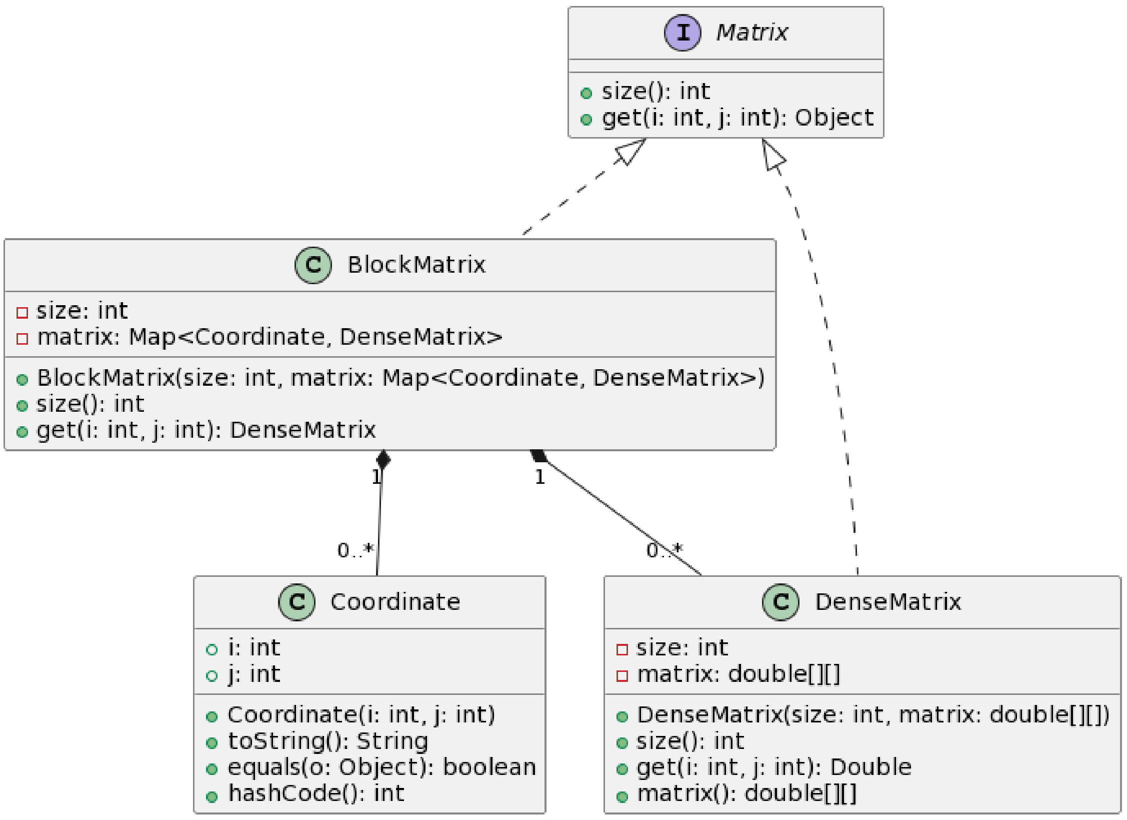

Matrix
size(): int
get(i: int, j: int): Object
BlockMatrix
size: int
matrix: Map<Coordinate, DenseMatrix>
BlockMatrix(size: int, matrix: Map<Coordinate, DenseMatrix>)
size(): int
get(i: int, j: int): DenseMatrix
1
1
0..*
0..*
Coordinate
i: int
j: int
Coordinate(i: int, j: int)
toString(): String
equals(o: Object): boolean
hashCode(): int
DenseMatrix
size: int
matrix: double[][]
DenseMatrix(size: int, matrix: double[][])
size(): int
get(i: int, j: int): Double
matrix(): double[][]

Figure A1: Module Model UML Diagram

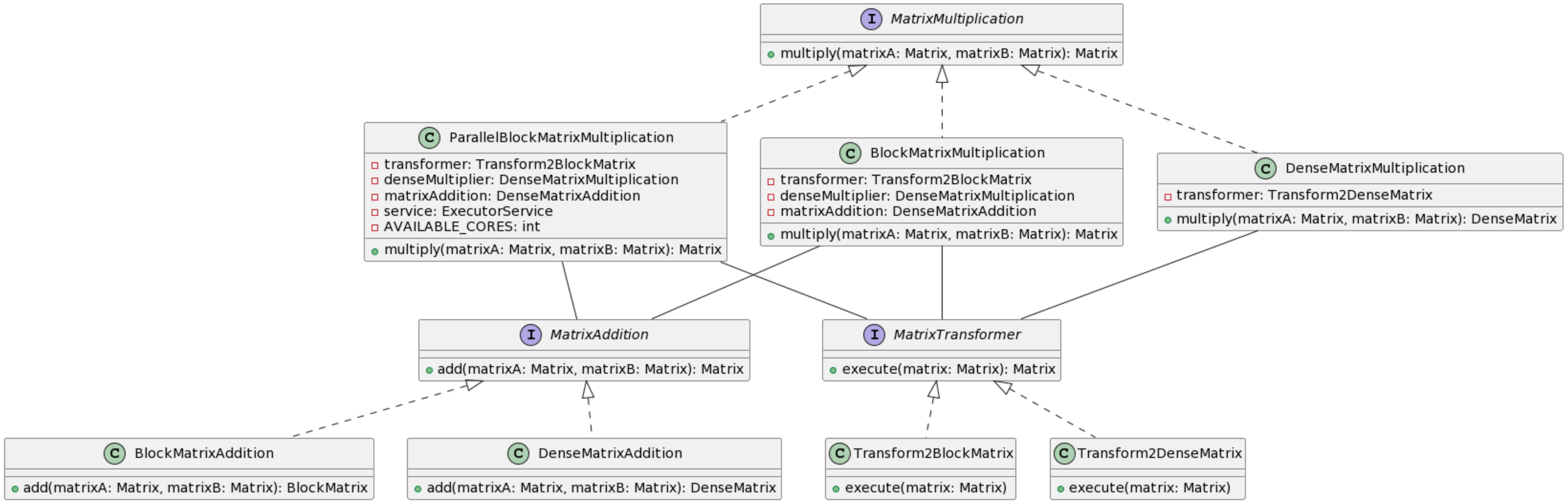
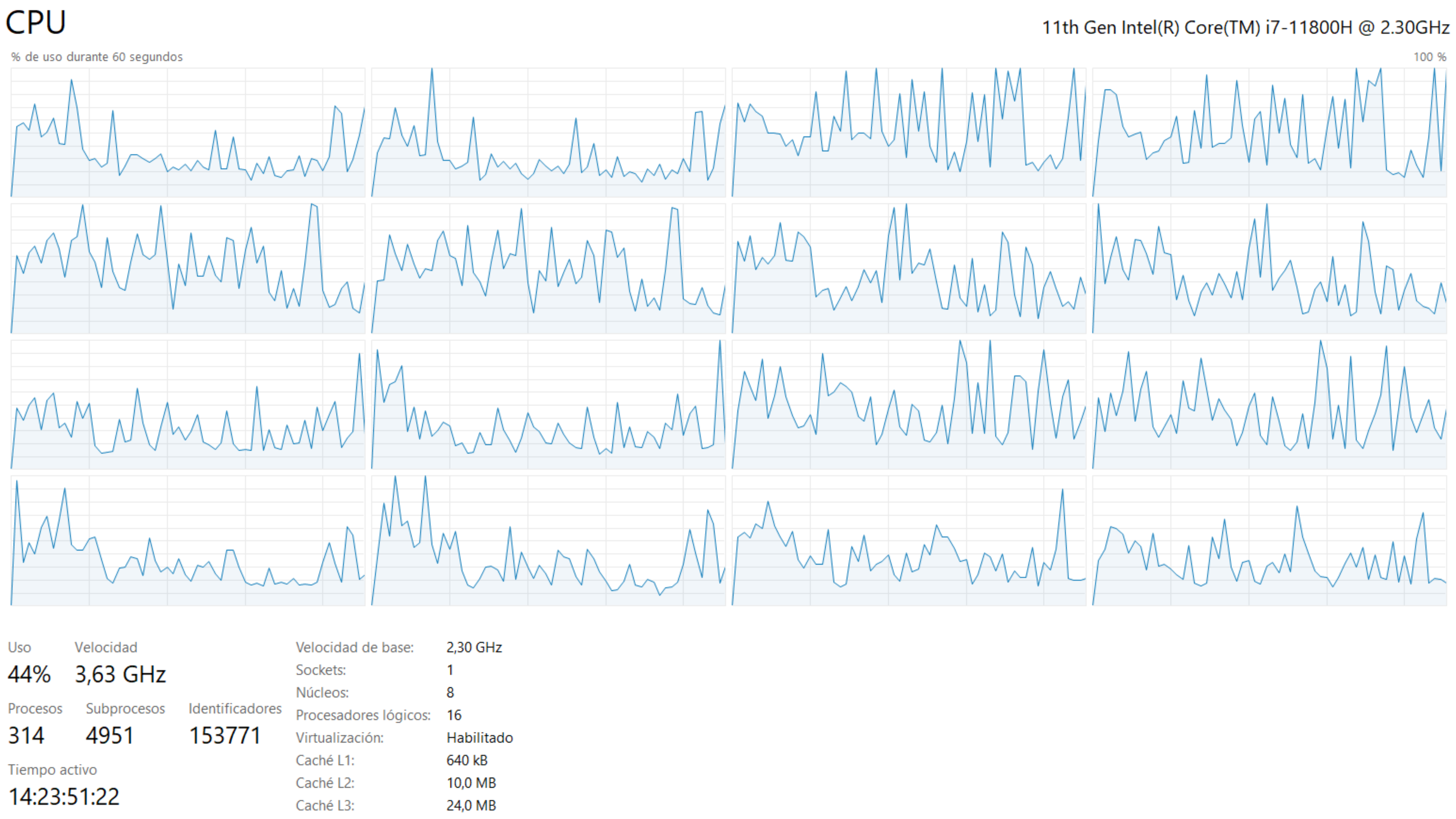

MatrixMultiplication
multiply(matrixA: Matrix, matrixB: Matrix): Matrix
ParallelBlockMatrixMultiplication
transformer: Transform2BlockMatrix
denseMultiplier: DenseMatrixMultiplication
matrixAddition: DenseMatrixAddition
service: ExecutorService
AVAILABLE_CORES: int
multiply(matrixA: Matrix, matrixB: Matrix): Matrix
BlockMatrixMultiplication
transformer: Transform2BlockMatrix
denseMultiplier: DenseMatrixMultiplication
matrixAddition: DenseMatrixAddition
multiply(matrixA: Matrix, matrixB: Matrix): Matrix
DenseMatrixMultiplication
transformer: Transform2DenseMatrix
multiply(matrixA: Matrix, matrixB: Matrix): DenseMatrix
MatrixAddition
add(matrixA: Matrix, matrixB: Matrix): Matrix
MatrixTransformer
execute(matrix: Matrix): Matrix
BlockMatrixAddition
add(matrixA: Matrix, matrixB: Matrix): BlockMatrix
DenseMatrixAddition
add(matrixA: Matrix, matrixB: Matrix): DenseMatrix
Transform2BlockMatrix
execute(matrix: Matrix)
Transform2DenseMatrix
execute(matrix: Matrix)

Figure A2: Module Model UML Diagram



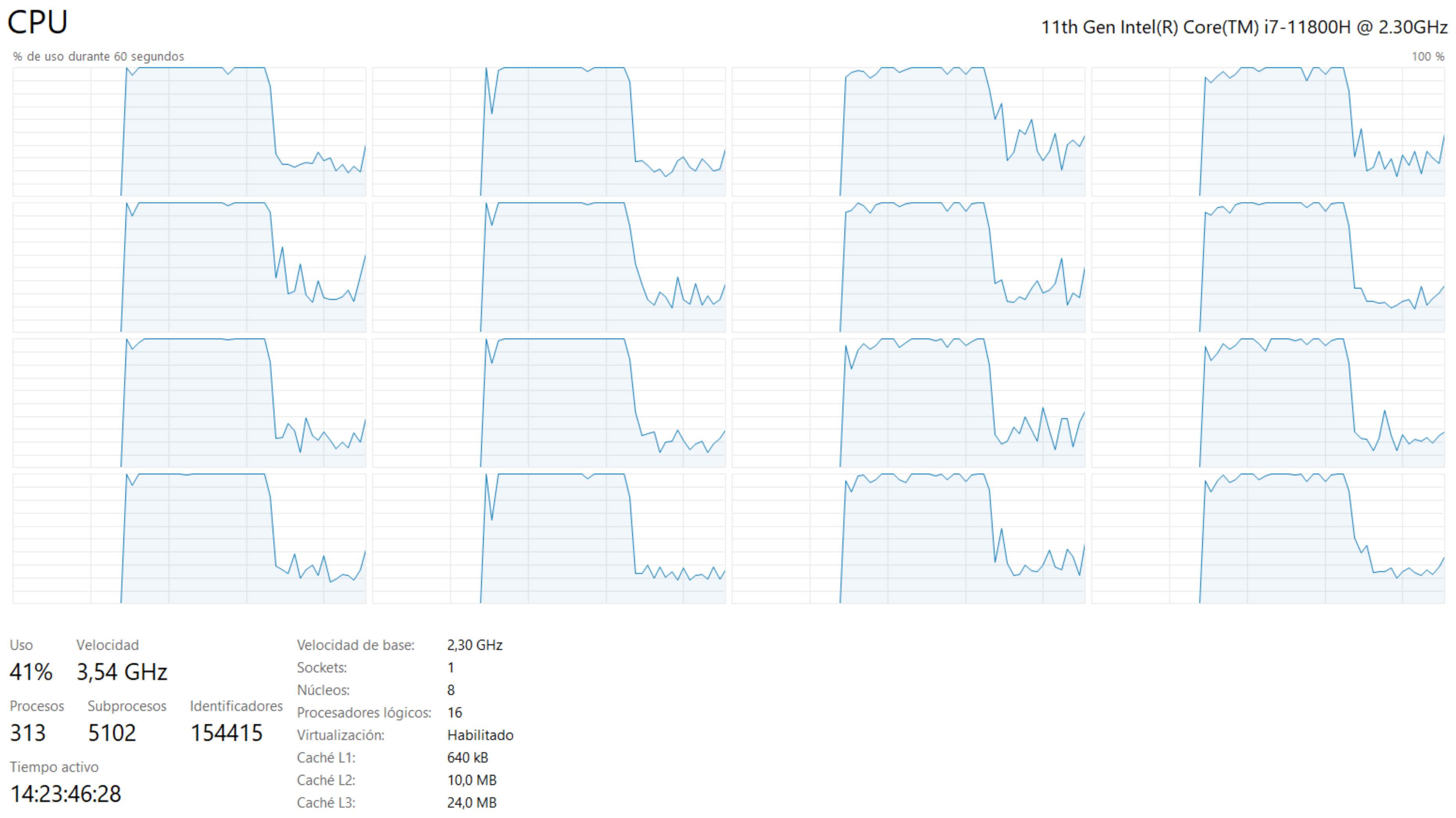
CPU

11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz

Uso	Velocidad	
44%	3,63 GHz	
Procesos	Subprocesos	Identificadores
314	4951	153771
Tiempo activo		
14:23:51:22		

Velocidad de base:	2,30 GHz
Sockets:	1
Núcleos:	8
Procesadores lógicos:	16
Virtualización:	Habilitado
Caché L1:	640 kB
Caché L2:	10,0 MB
Caché L3:	24,0 MB

Figure A3: Dense Multiplication CPU Distribution


CPU
11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz
% de uso durante 60 segundos
100 %

Uso 41%
Velocidad 3,54 GHz
Procesos 313
Subprocesos 5102
Identificadores 154415
Tiempo activo 14:23:46:28

Velocidad de base:	2,30 GHz
Sockets:	1
Núcleos:	8
Procesadores lógicos:	16
Virtualización:	Habilitado
Caché L1:	640 kB
Caché L2:	10,0 MB
Caché L3:	24,0 MB

Figure A4: Parallel Multiplication CPU Distribution