ULPGC · UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

BD

Parallelism and Synchronization: The path to execution optimization

Victoria Torres Rodríguez[1] [1]*victoria.torres101@alu.ulpgc.es*

Abstract

On some occasions, when creating a software product, the optimization of an algorithm can be so precise that the performance of the program does not depend entirely on it at some point in the development. Therefore, in this type of case, certain techniques focused on computational savings must be used: parallelism and synchronization.

The main experiments that have been carried out in this study have been the elaboration of a benchmarking on the different implementations based on these techniques (use of streams, threads, synchronization blocks...), measuring the time in milliseconds on the multiplication of matrices dense and sparse of dimension 1024 x 1024.

In addition, it has been wanted to focus on the use of these techniques to effectively verify that the execution time on the multiplication of matrices, both dense and sparse, is reduced by almost 50 times.

Therefore, it has been concluded that parallel programming shows great potential when it comes to obtaining the maximum benefit and computational savings, thus demonstrating that globally, the vast majority of programming cases are easily implementable with these methods.

Key words: *Execution time, Benchmarking, Parallelism, Sparse Matrix, Java, TDD, Matrix Multiplication, Synchronization, Threads, Semaphores*

1 Efficient exploitation of the division of tasks

Parallel programming can be defined in general terms as a technique focused on solving problems of great magnitude by dividing them into smaller sections that are solved in parallel. One of the most common approaches is matrix multiplication, a problem whose computational difficulty increases exponentially according to its dimensions (n x n). Taking this problem as the main focus, we must clarify that in order to meet the objectives of parallel programming, the use of two or more processors that can take care of the resolution of activities in the systems is required. It is also important to note that the design of a program with these computing parameters must take into account aspects such as the execution architecture, its space and time requirements, among others.

For example, this problem is being studied by numerous researchers, leading to the development of products such as Do! The project aims at the automatic generation of distributed code from multi-threaded Java programs. We provide a parallel programming model, integrated in a framework that restricts parallelism without any extension to the Java language (A Framework for Parallel Programming in Java, IRISA).

This paper analyzes the overheads incurred in the exploitation of loop-level parallelism using Java Threads and proposes some code transformations that minimize them. The transformations avoid the intensive use of Java Threads. All of these performing different benchmarking on the implementations that will be described later. In this way, we will realize that Java provides potential for this type of techniques in good use, they can develop quality software products.

2 Problem statement

In order to understand the importance of these techniques, the importance of concurrency must be understood. Concurrent programming is understood as a methodology that is implemented with the aim of solving problems

by executing different programming tasks. One of its characteristics is that it allows the system to continue with its activities without the need to start or end other tasks.

So concurrent programming includes the ability to reduce system response time by implementing just one processing unit. This implies that the task is divided into multiple parts that are processed simultaneously, but not at the same time.

In this case, you must define what the task unit is. Taking into account that we start from the multiplication of matrices, the global task is the calculation of said operation. This task can be divided into different subareas that are the calculations of each multiplication between the n rows and n columns of the matrix. It will be possible to verify later in the study that these subtasks will be treated as a critical section of the work. Once the main task and the subtasks have been defined, in this program we will start from this idea to take advantage of the maximum potential that these techniques can offer.

3 Method

3.1 Analysis and design

Focusing on the new aspects of the implementation, the most important aspects will be named.Firstly, these implementations are made up of individualized packages that have the appropriate classes for each technique. Secondly, it will be possible to verify that there are some implementations that have been developed for both sparse and dense matrices. This type of implementations have a common aspect, the use of Executor Service (simplifies asynchronous tasks by providing a pool of threads that are managed by this API, abstracting us from the work of creating and assigning tasks to these threads) and Streams ("wrappers" of collections of data that allow us to operate with these collections and make massive data processing fast and easy to read). This is due to the fact that this type of implementation is easy to adapt to any task unit, since they are usually concentrated in two general functions: a task to be carried out in parallel and a "concurrency handler" that will be the one that executes each task and maintains certain waiting times.

However, there are other less adaptable, but equally efficient, implementations that require manual concurrency handling, i.e. setting up synchronized semaphores, threads, and blocks. These three methods (with the exception of semaphores) have only been implemented on dense arrays. Semaphores can be defined in this work as a special variable that constitutes the classic method to restrict or allow access to shared resources.

At the same time, the difference between synchronized methods (semaphores and threads) and the synchronized blocks themselves should be clear in this study. One significant difference between method and synchronized block is that the synchronized block generally limits the scope of the lock. Lock scope is inversely proportional to performance, so it's always best to lock only the critical section of code. For synchronized methods, the lock achieved by the thread when it enters the method and when it releases when exiting the method, it either exits normally or throws an Exception. On the other hand, in the case of synchronized blocks, the thread blocks when it enters the block itself, and releases when it leaves the block.

3.2 Test environment

Once the changes made have been explained, we begin the implementation. The implementation is done in the Java programming language (version 11), an interpreted language widely used by companies around the world to build web applications, analyze data, automate operations, and create reliable and scalable business applications. The tests will be executed in the IntelliJ integrated development environment (IDE), specifically for the Java programming language, developed by the company JetBrains. * Please note that the tests will be run on a Legion 5 15IMH05, with a 2.60 GHz 2.59 GHz Intel(R) Core(TM) i7-10750H CPU and 16 GB RAM, so the test program Performance under other conditions will vary. The number of cores of this device is 6 and the number of available threads is 12.

The testing of the proper functioning of the classes has been carried out with different use cases, in the multiplication functions, checking mathematically (through the multiplication of a matrix by a random vector, multiplying it by A and B). All this has been executed under JMH, a Java harness for building, running, and analyzing nano/micro/milli/macro benchmarks written in Java and other languages targeting the JVM. All of them have been developed under the same circumstances and the same type of computer, described previously. It should be noted that much of the initial testing of matrices with the .mtx extension has been carried out with matrices provided from the paper "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms".

3.3 Methodology

Among the different types of multiplication methods that could exist, 5 have been defined, according to the different characteristics of the exposed techniques.

In the first place we define the methods by streams, both for dense and for sparse. When working with collections, our case arrays, we calculate and perform many types of operations on the data. Streams allow us through the functional paradigm to abstract from how to program those operations and only focus on what result is expected and write it in a very declarative way. In this type of implementation, we wanted to test with different numbers of streams. First of all, we have a single class with a single stream in the base matrix calculation algorithm (StreamDenseMatrixMultiplication) together with the use of lambda functions with forEach. On the other hand, we have this same algorithm implemented with 3 streams. The readability of the latter is notoriously better than in the former case, but its effectiveness will be compared in experimentation.

Secondly, we have the Executor Service API. Thread pools overcome this problem by keeping threads alive and reusing them. Any excess flowing tasks that the pool's threads can handle are held in a queue. Once either thread is released, they return to the next task in this queue. This task queue is essentially unlimited for the out-of-the-box runners provided by the JDK. All this replacing the traditional ways of creating threads in java (Threads extender or Runable extender). In our case it has been implemented in dense and sparse arrays, through a class with two main functions: multiply() and submit(). These are divided into the thread manager and the task to be performed.The implementation of semaphores is based on this idea, using them in the submit function of the Executor Service, both for dense and sparse.

In the case of the use of threads, we could briefly summarize it. A list of threads whose size is unknown at first glance has been used. We create a first loop with the length of the dense matrix, each time creating a new thread to which a task will be associated. Once said thread starts, it will be added to the previous list and there can only be a total of 12 active threads in the list in our case, because thanks to the Runtime.getRuntime().availableProcessors() function, it calculates from the number of cores of our computer, the optimal number of threads capable of performing tasks in parallel. If that list is full, it waits and no more threads are created until everyone finishes the task and the previously active threads are removed from the list.

The use of AtomicDoubles to store the results of the multiplication has been implemented, however its implementation is based on the use of the Executor service, so we will keep these implementations explanation as one from now on.The main purpose behind building atomic classes is to implement nonblocking data structures and their related infrastructure classes.Atomic variables also are used to achieve higher throughput, which means higher application server performance.

Finally, the implementation of block tones will be explained. The Java synchronized block is used to mark a method or block of code as synchronized. Java sound blocks are used to avoid contention. In our case it has been developed from an instance method. The connection of the Java instance method is synchronized in the object that owns the method. In this way, the method match of each instance is synchronized to a different object, that is, the instance to which the method belongs. Only one thread can be executed in the sound block of the instance method. If there are multiple instances, a thread can perform operations on a resonance block of instances at the same time. One thread per instance. For this reason, it has been decided to opt for a separation of responsibilities into 4 classes, a concurrency management, since this type of method is quite easy to have this type of error, so it will be in charge of verifying if a block is processed or not. . On the other hand, manual thread management will also be done, which will be in charge of assigning threads to each block and adding them to the task queue, until these blocks are processed and removed from the queue.

4 Experiments

The tests that were executed and the results that were obtained in said analysis will be demonstrated. To evaluate matrix multiplication implementation, it has to be must measured different aspects of performance in order to establish which one is the best.

These experiments have been developed under JMH, a Java harness for building, running, and analyzing nano/micro/milli/macro benchmarks written in Java and other languages targeting the JVM. All of them have been developed under the same circumstances and the same type of computer, described previously.

4.1 Experiment 1

First, a comparative study of dense matrices whose dimensionality increases in power of two up to a dimension of 1024 x 1024 has been carried out. We have divided the study into two parts: analysis of parallel and sequential methods and exhaustive study of each method. As can be seen in Figure I, we see that the vast majority of sequential methods maintain better times the smaller the dimension of the given matrices. Obviously this has a logical background, when it comes to generating threads, this process has a certain "expense" that in small

matrices exceeds even the minimum calculation time. Therefore, it should be noted that depending on the problem, one method or another must be chosen. It should be noted that the parallel methods are all under the same average behavior (with the exception of the transposed method), with a difference between methods of 500 milliseconds. Some notable points are that the implementation of a single stream takes 138.839 milliseconds with matrices of 1024 x 1024 while with three it takes more than double, 481,396 milliseconds. This occurs due to excess concurrency, as we discussed earlier. It is not proportional that a greater number of threads increases the speed of a process, since the handling of these is limited, and the greater this scope, the greater the delay generated by said concurrency. It is worth noting the notorious presence of a parallel method that deviates from the general norm, since it seems to double the execution times of the worst sequential methods.

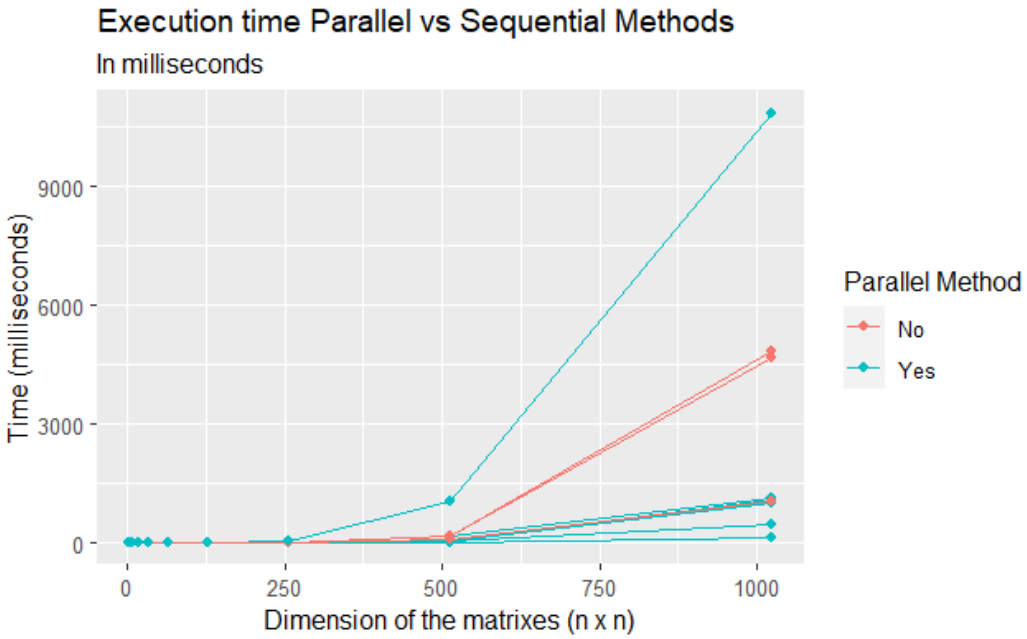

Figure 1: Execution time Parallel vs Sequential Methods

If we carry out a more exhaustive analysis, as can be seen in Figure 2, it will be verified that the best methods are those based on streams and those that make use of the executor service, since they are automated thread management formats, so their optimality has increased. been studied by java in depth, with the aim of finding effective and easy-to-implement methods. As could have been deduced, it is proven that the worst method is the use of atomics. This occurs because converting our dense array methods from double[ ][ ] to AtomicDouble[ ][ ] is costly. Instead of doing the setters like we do in our builder, we directly handle the array of atomics.

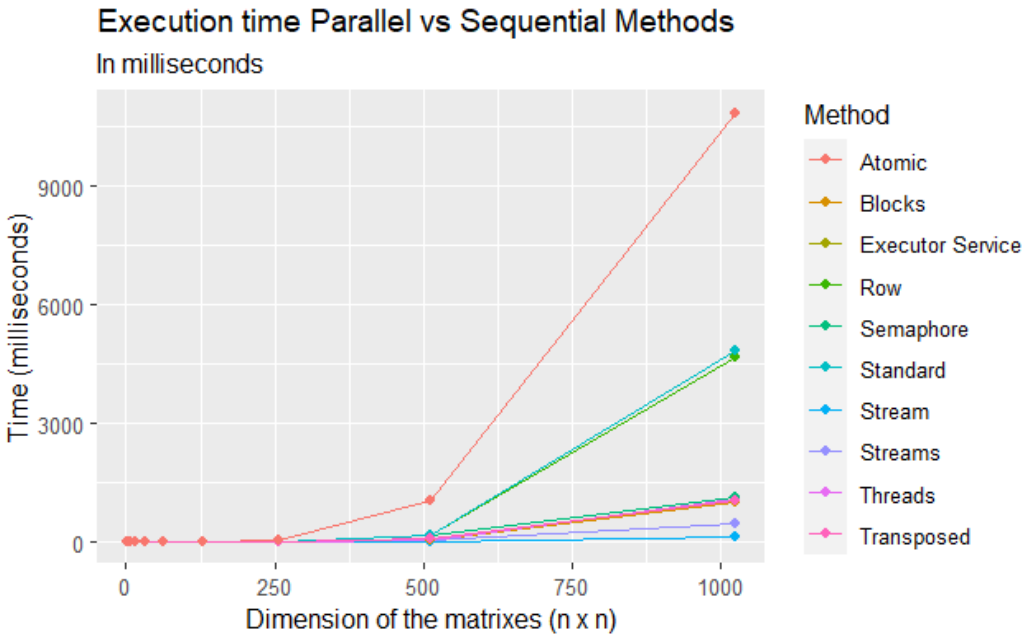

Figure 2: Execution time Parallel vs Sequential Methods

4.2 Experiment 2

In the same way as in the dense ones, a benchmarking has been carried out on the sparse matrices. As the general behavior is similar to that of the dense ones, the study will focus on the last dimensionality, that is, 1024 x 1024.

As can be seen in Table I, the response shown by sparse matrices, as has been shown in previous studies, is worse than that of dense matrices. Despite this, the same phenomenon is repeated, since the models based on executor service are the champion of execution times, these being 4 times faster, with an average of 1270 milliseconds per operation.

Method of matrix multiplication	Mode	Iterations	Execution Time	Units (ms/op)
SparseMatrixMultiplication	avgt	12	4,025.36	ms/op
ESSparseMatrixMultiplication	avgt	12	1,268.86	ms/op
StreamSparseMatrixMultiplication	avgt	12	1,965.35	ms/op
ParallelSemaphoreSparseMatrixMultiplication	avgt	12	2,637.81	ms/op

Table 1: Benchmarking of sparse matrix multiplication

4.3 Experiment 3

Finally, it has been decided to operate with larger size matrices, such as 500,00 x 500,000. These matrices come in default .mtx format, and their respective multiplications were performed both in dense and sparse form. It is not surprising that the dense matrices for this type of problem are slow, what is more, they give errors in their execution since the amount of memory they occupy greatly exceeds that of average computers. Therefore, its execution was unfeasible, giving memory errors. However, if we use the compressed formats we can see that although the execution is not excessively fast, it is efficient. The results obtained were

Other methods have also been tested in parallel that complement these methods. Again, parallel methods end up being the most effective method for problems that can be divided into a large number of subtasks. These methods end up being an average of 6 times faster (around 1200 seconds) with matrices of this high dimensionality, effectively executor service being the method par excellence in terms of speed, followed by the use of streams and threads and semaphores.

Type of matrix multiplication	Mode	Iterations	Execution Time (seconds)	Units (s/op)
BigSparseMatrixMultiplication	avgt	2	6,453.125	s/op
ESBigSparseMatrixMultiplication	avgt	2	1,245.875	s/op
StreamBigSparseMatrixMultiplication	avgt	2	1,683.632	s/op
SemaphoreBigSparseMatrixMultiplication	avgt	2	2,758.656	s/op

Table 2: Benchmarking of Big sparse matrix multiplication

5 Conclusion

The operations between matrixes constitute one of the bases of the functionality of many computer processes. Maintaining its optimal development can allow a software system to be maintained over time for a long time. As we have seen throughout this study, there are infinite ways to represent a problem in order to speed up its execution. In the first place, we have the sequential methods, those in which one task follows another, in our case, each multiplication of indices (i,j). It is a slow process where if one task is delayed the entire system must wait. The advantage is that it is easy to understand and implement. Secondly, there are parallel and synchronization (concurrency) methods. The main difference between these is that synchronization is a way of structuring a solution that can be parallelized (as is the case with synchronization blocks, which use threads, but structure the response differently).

Based on the benchmarking described above, it appears that transpose is an optimal way to perform matrix multiplications. Well, if it is about small matrices, whose dimensions do not exceed 500 columns and rows, they could be effective methods. However, as shown in experimentation, when dealing with a large matrix, we can see that performing multiplications with dense matrices without the use of parallel techniques is not a good option, since the execution time increases significantly. exponential. To reduce this coarse growth and try to alleviate the curve to a linear growth, parallel methods were used, which in their vast majority demonstrated to reduce up to 40 times said execution time (executor service, 130.56 milliseconds). This downward trend is replicated both in the compressed matrices (CRS and CCS), as well as in those whose dimensionality exceeds 500,000 x 500,000, being 6 times faster than the sequential methods.

For all this, what is the final conclusion? Parallelism is a very effective technique that allows you to significantly improve performance. However, the problem must be analyzed prior to the application of techniques, because in many cases if these are not divisible into subtasks, the cost of managing concurrency may be more "expensive" than the basic resolution of the problem.

6 Future Work

Many different adaptations, tests, and experiments have been left for the future due to lack of time (ie, experiments with real data are often time-consuming, requiring equal days to complete a single run). For all these reasons, we would like to name some aspects that would be taken into account for a better implementation of this study.

First of all, it is worth noting the closed record of parallel and synchronous solutions that have been demonstrated. In subsequent studies, new methods could be implemented such as those based on AtomicDoubles for example. In other words, we have used this tool, but we have not exploited its full potential, since we could have created a new matrix class based on this type of object with its corresponding builders, and thus analyze the full use of AtomicDoubles. Regarding the experimentation, the execution of each task could have been tested on different computers, to verify the resistance and optimality of each of the implemented methods, since it would be exposing a real case of problem solving. Second, the lack of matrix variability is notable, as this software product only handles double-type and square matrices. What's wrong with it? The lack of compatibility with future big data projects. Matrix operations are a foundation in algebra and many processes depend on this system. Because of this, we reduced the search for large-scale solutions. That is why the idea of creating an interface arises that, through the use of generics, allows us to edit and make any matrix format compatible, of the integer, double type... In addition, modifying certain aspects in the algorithms (such as how to check that the columns of matrix A are the same as the rows of B), we can treat multiplications with matrices of different dimensions, which are not square.

References

[1] *Clean Code* Robert C. Martin. 2019.

[2] *Multiplying Matrices Without Multiplying* David Blablock, John Guttag. 2021.

[3] *Sparse matrix storage format* Fethulah Smailbegovic, Georgi Nedeltchev Gaydadjiev. 2005.

[4] *Optimization of sparse matrix-vector multiplication on emerging multicore platforms* Samuel Williams†, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, James Demmel. 2007.