# Degradable Objects : An Application To A Distributed Data Store

Adam CHADER
*PDS track, Master of Computer Science*
*Institut Polytechnique de Paris*
Palaiseau, France
adam.chader@telecom-paris.fr

***Abstract*—Ceci est un abstract**

***Index Terms*—Concurrent programming, parallel programming, performance, databases**

## I. Introduction

This paper is the report of a research project realised for the Parallel And Distributed Systems master at *Institut Polytechnique de Paris*. It describes the work that I realised to apply degradable data structures to Apache Ignite [2], a distributed data store, to improve its performance and scalability.

### A. Context

Up until the begnining of the 21[st] Century, there had existed a simple rule that could be used to predict the performances of computers. This is known as Moore's Law (Fig. 1). This law states that every two year, the number of transistors would double on chips. This rule could be used quite flawlessly to predict the speed of software, as there is a direct correlation between the speed of sequential programs and the frequency of CPUs. However, in recent years, several issues have surfaced regarding these assumptions : Firstly, we seem to have reached the minimum physical size of a transistor at around 5 nanometer ; secondly, the frequency that can be reached by craming all these transistors in a single CPU core is too high to be dissipated by current means [4]. As of today, it is extremely complicated and costly to significantly increase the clock speed of a single core. The solution that was found is to put multiple CPU cores in a single machine. The total number of clock cycle per second should continue increasing, while the heat would remain manageable. Intuitively, this solution is perfect : by having two cores instead of one, we could double the total frequency of the CPU, and thus double the speed of software. This is what we call *linear scalability*, and it is still just a dream. There are many reason as to why it is impossible to achieve linear scalability, but they all boil down to the same issue : in order to make the most out of several CPU cores, we need to have them working at the same time. It is possible when the tasks are perfectly parallelisable, but this almost never happens when we want the cores working together. Programs need to access variables in memory, and accessing them in parallel can cause conflicts (we will explain these conflicts in detail later), they also need to aces IO which can cause an issue. The resulting effect of these issues is that parallel programs often contain sequential portions, where each core has to wait for one core to finish a task. The more sequential portions a parallel program contains, the slower the speedup is. This is described by Amdahl's Law [5] (Fig. 2). It describes the speedup of parallelisation as a function of the amount of sequential code.

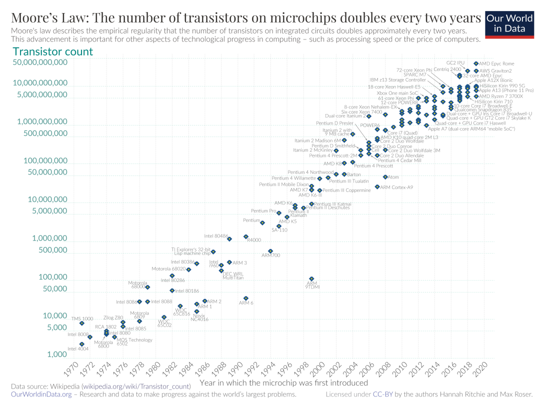


Fig. 1: Moore's Law.

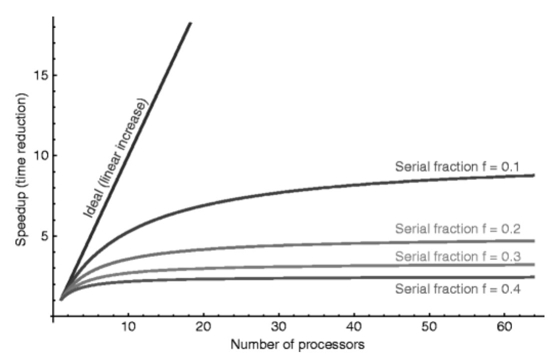


Fig. 2: Amdahl's Law

With all this in mind, it becomes clear that the burden of increasing the performances of software no longer lies in the hand of hardware manufacturers, but rather on the hands of software engineers, who should attempt to have as little sequential code as they can in their programs. Historically, it was the responsibility of the end programmer to parallelise their program. But there has been work to hide this work in the lower levels of the software stack [1], to make the shift from sequential to parrallel transparent to the higher levels. The scope of this work is at a lower level. The goal is to provide perfectly linearly scalable implementations of classic data structures, keeping the exact same interfaces, such that there could be virtually no modification to applications' source, while achieving the scalability that we desire.

### B. Work

This project is a part of a bigger project around *Degradable Data Structures*. These structures will be explained in more detail further in this paper. Simply put, these data structures are created by tweeking the specifications of classical data structures, in order to be able to change their implementations to better ones allowing for better scalability or performance. The idea is to take real industrial code, and change some of the objects used for degraded ones whithout having to change any of the code, and observe the increase in performance. The goal of this specific project is to apply these data structures to a specific use case being the data store Apache Ignite [2], and observe the evolution of performances.

### C. Contributions

This project made the following contributions to Apache Ignite:

- Benchmarking
- Perfomance Analysis
- Analysis of the source to find possible applications of degradable objects

### D. Outline

The rest of this paper details to context and the contributions of the project.

We will first explain the theory behind degradable objects and how we can find places to apply them.

We next will detail the information we gathered by analysing Apache Ignite.

we will then show the result obtained.

## II. Related Work

### A. The Scalable Commutativity rule

The approach of Austin Clements [1] team is very similar to the work being done here. They present a new way of designing scalable and performant software, by thinking on the interface rather than on the implementation. They present the *Scalable Commutativity Rule* that states that *whenever interface operations commute, they can be implemented in a way that scales*. To describe this commutativity they provide a new definition calles *SIM-Commutativity* for more complex interfaces. The idea is then to modify the interfaces so that the operations are commutative. This relates very much to the idea of degraded data structures, as they are an extension of this work : rather than saying that the interface needs to be modified, the idea with degradation is to remove some constraints of the previous interface, so that it scales. It could use commutativity or some other form of contraint on the interface.

In the scalable commutativity rule, they work on the interface of the linux kernel, whereas this work is more geared towards data stores and their data structures.

## III. Degradable Data Structures

The goal of this project is to improve the performances of Apache Ignite, by inserting existing degraded data structures in the place of traditional data structures. We wil therefore first explain to what extent the implementation of data structures is linked with performance of parallel applications. We shall then explain the principle behind the degrading of objects, and we will finally explain how we can find the places where it makes sense to apply this degradation.

### A. Contention and Scalability in data structures

The relationship between the implementation of data structures and the performances of parallel applications is not immediate.

To achieve the best performance possible with parallel software, we need for it to scale. Meaning that increasing the number of parallel workers will increase the speed of the computation propotionally. As we have seen with Amdahl's Law, to do this, we need to have as little sequential code as possible in our program. One big source of sequentialness in software is contention is data access. Indeed, when two process try to access and modify the same data point at the same time, the result is non deterministic, which is really bad. To prevent this, safety measures are put in place to prevent concurrent accesses to the same piece of data. Often times, this results in the fact that only one process at a time can access the data, which we will call a **bottleneck**.

We can therefore conclude that the goal of the designer of data structures is to prevent bottlenecks by avoiding contention on the same data points as much as possible

### B. What are Degradable Data Structures ?

### C. How do we find where to apply degradation ?

## IV. Performance Analysis of Apache Ignite

## V. Performance Evaluation

## References


[1] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. 2015. The scalable commutativity rule: Designing scalable software for multicore processors. ACM Trans. Comput. Syst. 32, 4, Article 10 (January 2015), 47 pages.

[2] Apache Ignite

[3] Google Cloud Platform Compute Engine

[4] R. R. Schaller, "Moore's law: past, present and future," in IEEE Spectrum, vol. 34, no. 6, pp. 52-59, June 1997, doi: 10.1109/6.591665.

[5] Gene M. Amdahl (1989), Oral history interview with Gene M. Amdahl, Charles Babbage Institute, University of Minnesota, hdl:11299/104341.