Parallelization in String Matching

André Figueira
m37280@alunos.uevora.pt

Universidade de Évora, Departamento de Informática

January 25, 2017

Abstract

This project implemented in Golang means the Language and learn Parallelization mostly in the area of String Matching and to understand its uses and benefits over traditional sequential means of dealing with problems. This was done by partitioning the text by cores or total workload or workload per core and then performing a serie of tests where file size increases comparing to sequential problem solving of the same task. Results show a significant difference between sequential and Parallel computing where it showed a significant reduction in string matching time.

keywords: String; Matching; Sequential; Parallelism; Performance; Go

1 Introduction

Sequential computing is an ordered and consecutive execution of processes, one after another. Most standard algorithms are sequential this due because in previous time, new generation of individual cores were getting higher and higher clock speed and therefore the need for parallelization wasn't being justified because a single core with a much higher clock speed would have a significant performance over weaker multiple cores.

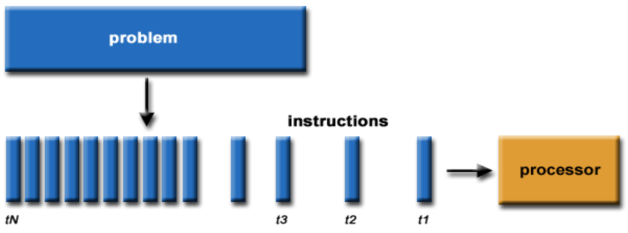

Figure 1: Sequential problem solving.

Figure 1 represents sequential problem solving, in which the problem is divided into instructions, and those are executed sequential manner, that is one after another. But in terms of today the generations of CPUs are getting closer to their physical limit meaning that sequential problem solving won't get any faster and cant always be the solution to the problem, hence **parallelization**.

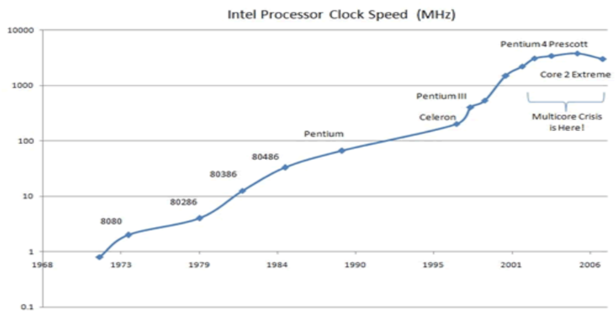

Figure 2: Intel CPUs clock speed evolution over years.

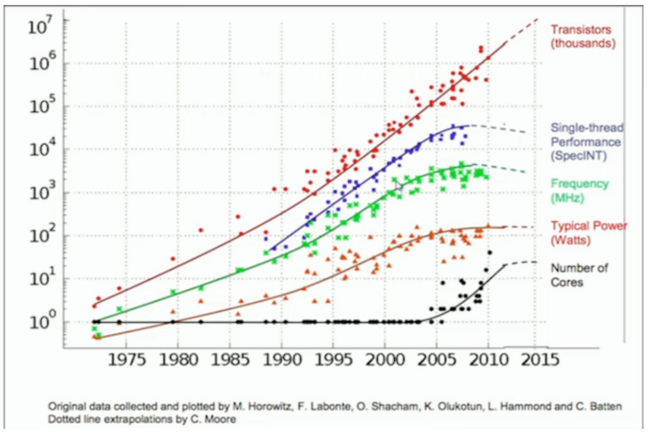

Figure 3: Evolution of transistors, frequency, performance and number of cores.

Figure 3 shown represent the evolution of clock speed as new generations of CPUs come to existence. It can be easily seen that clock speed starts to change from a linear progression to a quadratic progression since the last decade, meaning that as stated before the physical limitations are visible, and increasing the number of cores of a CPU becomes a new solution, meaning that **parallel computation starts to gain popularity over sequential computation**.

Sequential computing shows an effect on String Matching. The file to be searched will have an impact on the overall performance, if the file size scales so will the time the pattern to be searched, as a result will affect the performance, making parallelization of the task a reality.

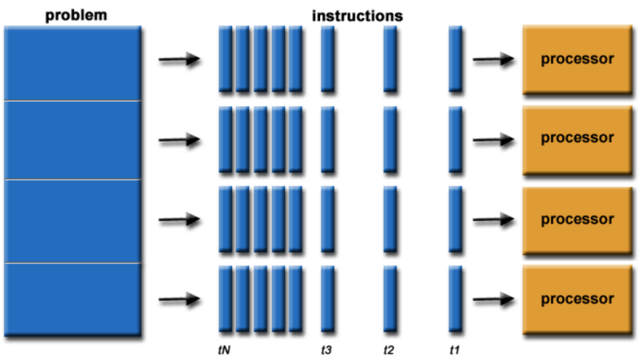

Figure 4: Parallelism problem solving.

Figure 4 represents **Parallelization**. Is the act of which multiple computations done in a machine are not in sequential form but in parallel, meaning multiple computations which are independent from other computations, can be done at the same time without waiting for the previous one to finish, taking advantage of the multiple cores in a machine CPU.

In regards of **String Matching** where data dependence is minimal makes it an ideal candidate for **Parallelization, where performance and speedup**

is key to diminish the overall time and cost.

So in order to clarify sequential vs parallelization a number of of tests with numerous files of different sizes with increasing number of Threads and cores to increase parallelization possibilities were conducted to show gains and losses in speedup for both sequential and parallel computations, and conclude whether parallelization is an optimal solution for String Matching problems. This comparison will help to understand the benefits of parallelization over sequential computing.

Test 1 will check the performance of only one thread per core.
Test 2 will check the performance of increasing workload. Meaning multiples threads per core.
Test 3 will check the performance of total workload with increasing available cores.

2 Architecture

In String Matching architecture, it is required to select a file and afterwards a pattern to be searched. The objective for such could be the number of occurrences or find whether a certain pattern exists in a given file, for given reason, for example. The issue with this architecture is that for longer searches as in files with increased size performance becomes an issue, and could be taking considerably amount of CPU time to fulfill a given task.

The Figure 5 represents the sequential architecture. This is the basis of the comparison between sequential and parallelization architectures. The architecture for the sequential part is the following:

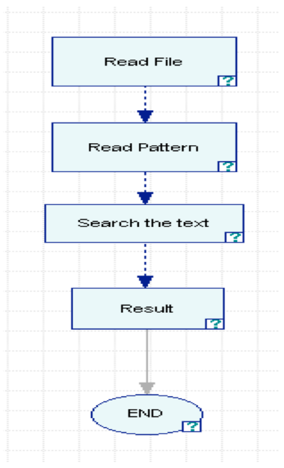

Figure 5: Sequential Architecture.

For the second part of the problem at hand, another architecture was developed to try and get to achieve more interesting and better results. Parallelization of String Matching, for that it was proposed the following architecture.

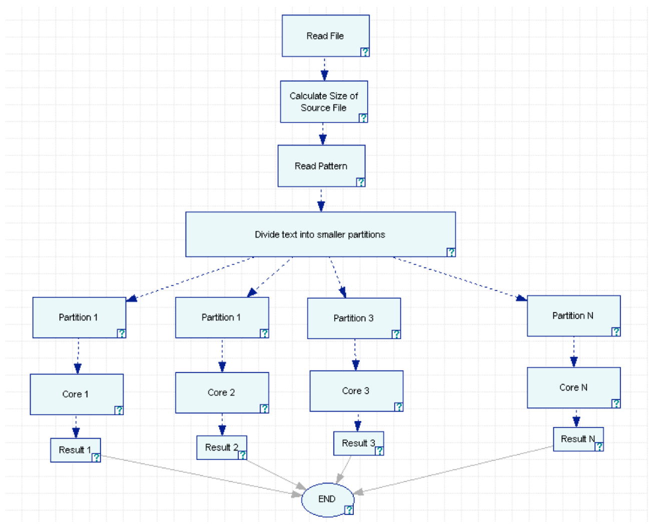

Figure 6: Parallel Architecture.

The following architecture uses only the virtual cores of the machine, and is limited to the amount of virtual cores as the machine has. So it does not take advantage of the physical cores as such is the properties of the programming language [Golang] used in the implementation of this architecture. So the parallelization was done taken only into account the virtual cores.

First step is to implement the architecture stated in Figure 6 and the architecture stated in Figure 5.

It works in the same way as the sequential architecture up to the read pattern. After that, the file is divided into smaller partitions and each partition is dealt with a corresponding core, calculate the result. There are cores that may finish faster than others so a WaitGroup was created so that the cores will wait for others to finish and then add up the results.

This is taken into consideration that a core gets only one workload, that is, a core runs one partition and so on. Based on that architecture three tests were conducted. The first one is exactly what what is represented in the architecture. Another test conducted and that was workload per core, similar to the previous test but for example, 8 cores, 16 workload, it means the text would be divided into 128 partitions. The final is total workload per available cores, that means, for example, 2 cores, 8 workload, it would mean that the text would be divided into 8 partitions.

These partitions are computed in threads, or lightweight threads known as Goroutines. And are computed as soon as there is availability from the cores.

These tests are all in comparison to sequential architecture. And comparison between workload. The reason behind these two final tests is cores since they can finish their partitions faster than others depending on the type of partitions, and instead of becoming idle, they will pick up idle threads (if there are any) and parallelize them, increasing effiency.

3 Tests

In this section the results of the tests will be posted with corresponding graphs so that observations can be made.

The language used in the implementation of this system was Go Programming Language or Go/Golang for short. Which permits the programmer to control the amount of virtual cores (limited to the machine virtual cores) a program uses.

3.1 Test 1

This test was using only one thread (or Goroutine as in Golang) in each core for different file sizes. These tests were in comparison with sequential time using only one core.

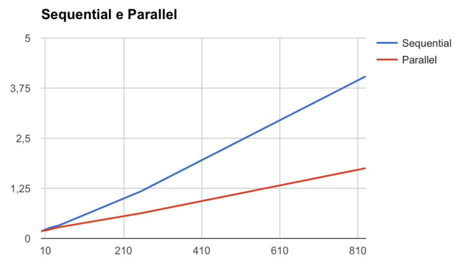

Figure 7: Sequential vs Parallel.

Figure 7 shows the difference between sequential times and parallel times as file size increases, the difference is noticeable, parallel is outperforming sequential as expected. However is file sizes below 10mb the difference is minimal having similar values. But as the file size increases the difference becomes very noticeable showing a very big difference. The speedup graph is as shown:

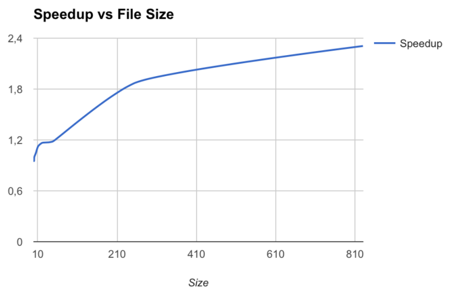

Figure 8: Speedup with file size increase.

As expected the speedup is increasing with file size. With low file size, not showing a significant improvement. This shows that parallelization has a slight improvement but can only really be visible with larger files. The following graph shows differences between using multiple virtual cores, from two cores to the maximum the machine has (8 was the maximum of testing machine).

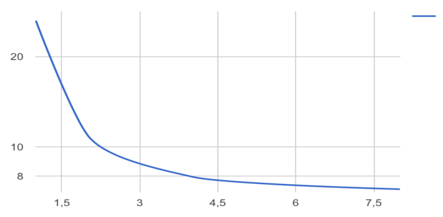

Figure 9: Available cores vs time.

Figure 9 is the difference of time and increasing multiple cores using a file with 3,32gb searching for the pattern 'GAT'. It shows a significant difference from 1 to 8 virtual cores, but not such a difference from 4 to 8 cores only differentiating about one second. But sequential (one core) and 8 cores the difference is significant compensating the use of parallelization over sequential computing.

3.2 Test 2

This test was taken into consideration the increasing workload of cores, meaning that a file would be divided into smaller partitions and each thread would be associated with a partition of the original text. Similar to the previous test (test 1) architecture, but each core will perform more than one thread per core.

With one workload per core, it is the same situation from the previous test.

The following graphs will show increasing workload and comparisons of time and file size. Same environment as the previous test.

The reason is to hopefully get more interesting results than the previous test since cores can finish faster than others (if partition is easier to deal with) and instead of becoming idle, they can pick up waiting threads.

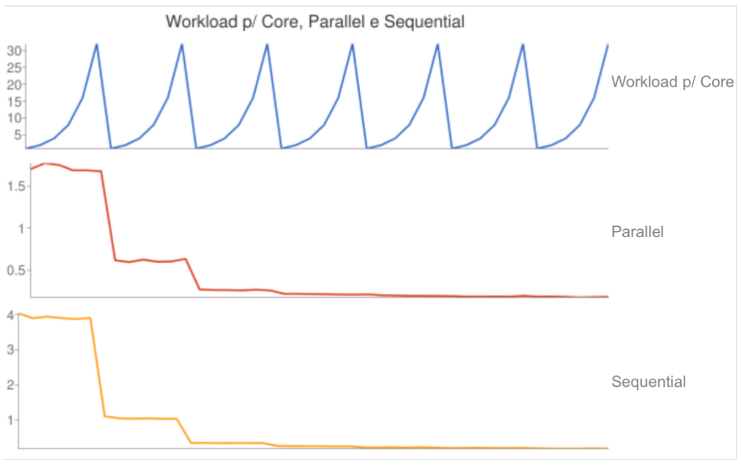

Figure 10: Sequential vs Parallel vs Workload per core.

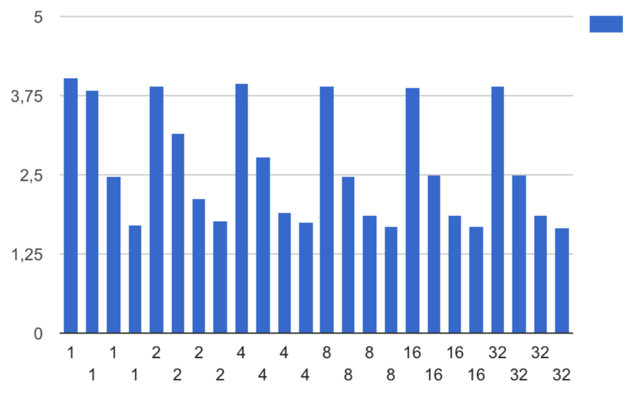

Figure 11: Workload increase vs time with a file of 829,9mb.

By observation of the bar graphs it shows that significant differences in time reduction. But in some cases minor increase of time with more workload, although the conclusion is of others factors that could affect the machine, so the conclusion is it makes no difference.

Since the previous results were interesting, it was decided to another test but with a file of 3,32 gb of size.

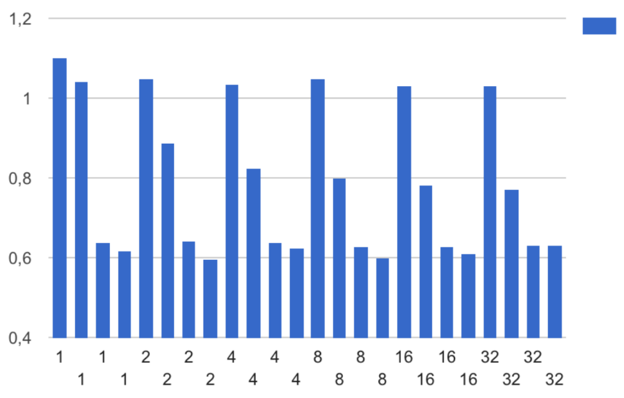

Figure 12: Workload increase vs time with a file of 253,9mb.

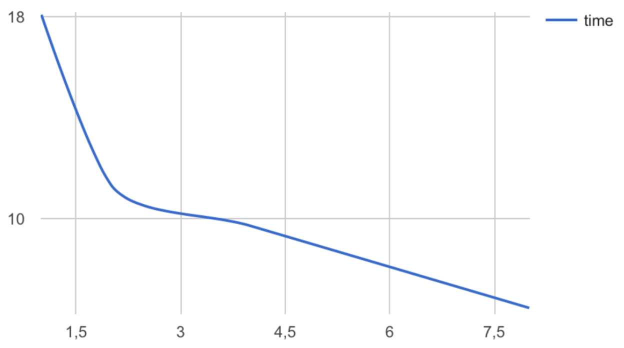

Figure 13: workload of 32 vs time.

In this graph the results are more interesting. The increased size in file showed a clear difference between cores with a 32 workload per core, showing a much better performance when using 32 threads per core with 8 cores available.

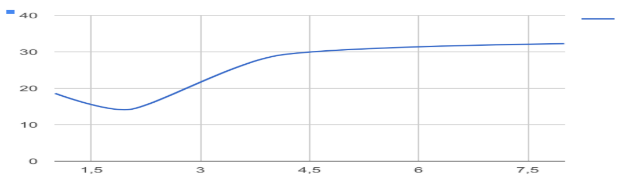

Figure 14: workload of 32000, time vs available cores on a file with 3,32gb.

Figure 14 represents the situation "how many threads is to many", meaning that more workload does not mean exactly better performance, and in this case, with 320000 threads per core the performance got worse with increasing available cores.

3.3 Test 3

This test was to see given the same workload in general and the available cores what results could happen. This means for example, 8 cores, 32 partitions. Meaning there are 8 available cores to process 32 partitions.

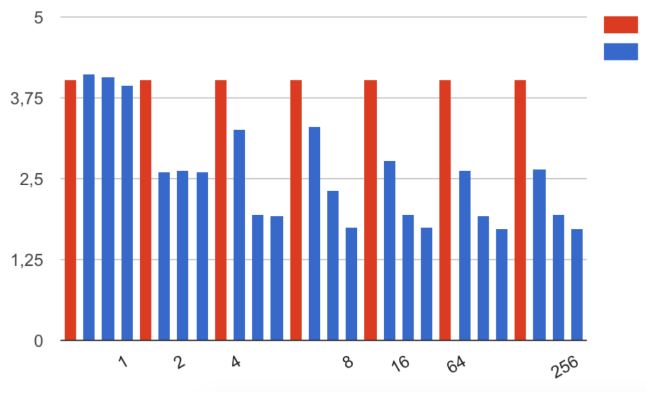

Figure 15: Sequential vs Parallel with varying total workload with file of 829,9 mb.

Figure 12 the red Bar is the sequential time, the blue bars and the parallel time 2, 4 and 8 cores, respectively. Time tends to decline with number of available cores as expected because their are more options for parallelism.

The next graph is of workload of 320000 in total on a file of 3,32 gb

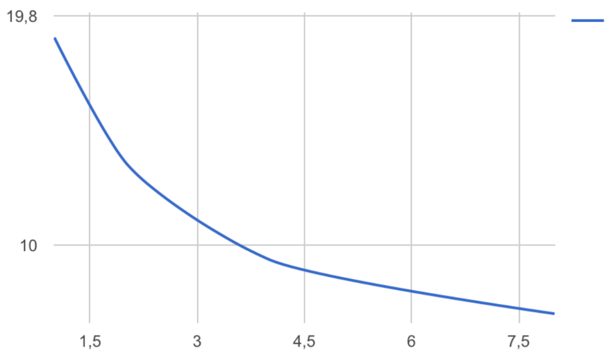

Figure 16: workload of 320000 with increasing available cores on a file of 3,32gb.

In larger files the performance is clearly visible, as there are more available cores the more parallelization can occur.

4 Comparing Related Work

Articles in this area are more inclined to a specific task such as analysis of a Genome and patterns, for example, and using multiple more complex algorithms and seeing differences in speedup with those algorithms.

It can be said that related articles mostly concluded the same although using different approaches.

"This Parallelization greatly improves the matching efficiency if the text size is very large and a sufficient numbers of processors are available" - A Parallel Computational Approach for String Matching - A Novel Structure with Omega Model By K Butchi Raju and Dr. S. Viswanadha Raju

"The simulation results show that the performance of string matching algorithms namely execution-time and speedup improved" - Parallel String Matching Algorithm Using Grid by K.M.M Rajashekharaiah, Ch MadhuBabu and Dr. S. Viswanadha Raju

These articles proceed with the same manner as this one. With the difference, they used multiple algorithms and dealt with the problem with for different reasons, and used multiple algorithms. While this one, did not use multiple algorithms (only one) but in the end the conclusion was the same, to see if parallelization would show increase performance and speedup to the given task at hand.

5 Conclusion

The aim of this project was to learn more about parallelization and its capabilities and flaws and the language Golang used to implement this project. By learning parallelization in the area of String Matching Algorithms and learn why should it be used and when not to and using the sequential architecture as a baseline of comparison.

5.1 Sequential vs Parallel

After the tests were performed the results were interesting. There were significant differences as shown by the graphs. But more workload meant increased performance and therefore speedup although with increased number of threads it seemed to stall and only vary slightly. This for only virtual cores.

For the objective of sequential vs parallelization it's clearly visible the differences, as with much larger files (highest tested being 3,32gb) the difference is very significant showing a good speedup. For small files (lowest tested was 1 mb) the differences were not so visible, it didn't compensate as much as it would for larger files, as the speedup would be slightly be worse because the effort to force parallelization would cause a worse performance.

Although with a colossal amount of workload in a large file it hurt the performance quite severely because it was to many threads for the CPU to handle for the particular task at hand, meaning that for optimal performance, there must be a value in between to get correct amount of workload per core without hurting the speedup.

The overall objective was to learn about Parallelism and Golang, which could be said that was achieved.

5.2 Overcoming limitations

Since Golang has its limitations when it comes to parallelization, that is the only being able to take advantage of the virtual cores of a machine. The next step would to re-create the architectures shown previously but on a different language, one that would take advantage of the physical cores in a machine and see what results could be taken out of performing tests on String Matching.

5.3 GPU?

Since nowadays having a dedicated GPU on machine is a very common thing, the next step would be re-create the architectures shown previously but on a language that would take advantage of machine GPU and see what gains or losses would be achieved when performing tests of String Matching.

5.4 GPUs?

Nowadays a machine with one dedicated GPU is very common. But having two dedicated GPUs is not so common. Although in areas like computer gaming, two GPUs is usually seen on high end computers or laptops for high performance gaming. So the proposed work would be to try and take advantage of the multiple GPUs and see what interesting results could be retrieved when performing tests.

References

[1] *Parallel Computational Approach for String Matching - A Novel Structure with Omega Model By K Butchi Raju & Dr. S. Viswanadha Raju* `https://globaljournals.org/GJCST_Volume13/3-A-Parallel-Computational.pdf`

[2] *Parallel String Matching Algorithm Using Grid by K.M.M Rajashekharaiah, Ch MadhuBabu and Dr. S. Viswanadha Raju* `http://airccse.org/journal/ijdps/papers/0512ijdps03.pdf`

[3] *Pattern Searching in a Single Genome Abigail Linton* `https://www.stats.ox.ac.uk/__data/assets/pdf_file/0018/5373/LintonFinalReport.pdf`

[4] *Introduction to Parallel Computing* `https://computing.llnl.gov/tutorials/parallel_comp/#Whatis`

[5] *Wikipedia. What is Parallel Computing* `https://en.wikipedia.org/wiki/Parallel_computing`

[6] *Concurrency, Goroutines and GOMAXPROCS* `https://www.goinggo.net/2014/01/concurrency-goroutines-and-gomaxprocs.html`

[7] *Pool Go Routines To Process Task Oriented Work* `https://www.goinggo.net/2013/09/pool-go-routines-to-process-task.html`

[8] *Golang* `https://golang.org/doc/`

[9] *Parallel Implementation for String Matching Problems on a Cluster of Distribuited Workstations* `http://users.uom.gr/~pmichailidis/jpapers/jpaper003.pdf`