My Army: Strategy Game Engine

Alexandre Freitas
alexandre.freitas@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November 2015

Abstract

This document explains all the process used to remake the game engine of the "*My Army*" game. The process starts with a detailed review to the game engine to understand how it works and it was also made a review between the current language and the new language to understand the capacity of each language to solve the existing issues. Then, a solution was presented to solve the existing issues, according to the previous reviews. After weeks of development and solving issues, a new version of the simulator was complete. The next step was to analyze this version to insert parallelism to improve the simulator performance. After several attempts, a parallel solution was reached which offered a better performance and consistent battle outputs. Finally, all the battle simulator versions were tested using different battle examples from the game in order to make a comparative analysis showing the performance from each version.
Keywords: game engine, battles, performance issues, simulator, agents

1. Introduction

Every game created till today started as a simple idea and to turn this simple idea into reality is not as easy as it sounds because it requires to go through a long and time-consuming process to get the right resources and also to refine as much as possible this idea.

Following the desire to have his own game, there was a person who had the idea to create a new strategy game, whose name was "My Army", in which the players could own and manage an army somewhat similar to what a general would do in real life.

This game had a performance issue in one of the modules. This module was an agent-based battle simulator responsible for the battle management where each battle can have a large number of agents. Beyond the JavaScript version there was also an incomplete version developed in Go.

The main challenges were to study the existing version of this module, to complete the unfinished version developed in Go and to solve the performance issue by using a new high performance programming language and using a parallelism approach.

2. Background

2.1. Game Analysis

"My Army" allows the players to be on a role of a general giving them the authority to own and manage an army to battle other players. The army management is done by recruting troops with different characteristics, purchasing different equipments to improve the soldiers' performance and creating several tactics in order to achieve the game's main goal, which is to win as many battles as possible to reach the top of the ladder.

2.1.1 Battle Simulation

The main focus was the module dedicated to the battle simulation, which was developed in JavaScript and managed all the elements from a battle between two generals. When a general challenged another one, this module received all the information about the two generals and generated a battle composed with several agents.

The simulator creates a rectangular battlefield where each side is assigned to a general to put his army in the battlefield according to his tactics.

The tactic created by a general sets the army formation and the army behaviour, both offen-

sive and defensive, in the battlefield. The army formation can have one or more lines of combat where each represents the alignment of a group of soldiers at the beginning of each battle. [8, p. 31].

There are three different sectors in a battlefield: a central sector, a left flank sector and a right flank sector. These three sectors have a set of rules that may vary from sector to sector. [8, p. 31].

Each sector is divided in slots where each of these contains a category and a relative power (Strong, Average, Weak).

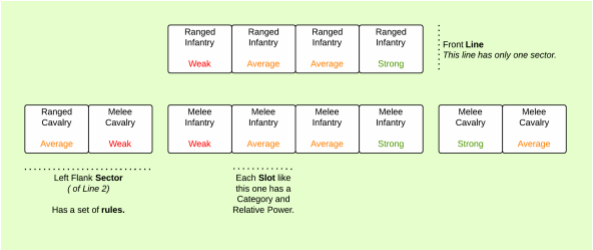

Figure 1: Example of a tactic with two lines of combat. [8, p. 31].

In general, the melee soldiers are specialized in close combat while the ranged soldiers are specialized in ranged combat using ranged weapons and avoiding at all costs any kind of close combat. On the other hand, there is the artillery which uses high damage equipments but they have low mobility or no mobility at all.

The simulator uses all the previous information about the tactic and the player soldiers to divide the soldiers in battle contingents and put them in their initial position according to the player's tactics where each battle contingent is represented by a rectangle whose size varies with the number of soldiers in it and this means that if the number of soldiers decreases the size of the rectangle decreases as well. The rectangles cannot overlap other rectangles at any time during the battle except when an allied battle contingent is running from the battlefield.

Each battle contingent is an autonomous agent with a *Finite State Machine* that is responsible to decide what type of behaviour the agent must have along the battle. These agents have different possible behaviours such as moving towards an enemy or a spot in the battlefield, being able to rotate when needed, selecting a weapon to use depending on the distance to the enemy, selecting new targets to attack and helping allies in danger. Each of these actions can be slightly different between agents because it depends on the actual given order. They are also considered as ominiscient agents, this means that they have full vision of the battlefield, knowing all the enemies positions. [8, p. 49].

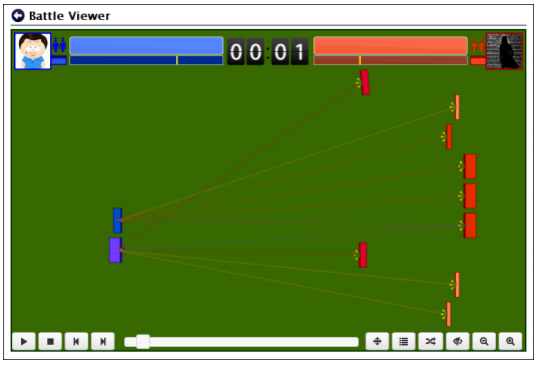

Figure 2: *Battle Viewer* [8, p. 50].

Each contingent has an orders list that is established during the creation of the tactics and during the battle which will affect its behaviour throughout the battle. The contingents will always try to fulfill their current order even though they are in a bad spot.

This orders list has different types of orders that lead to simple behaviours, such as melee attack, ranged attack, defend, pursuit, and complex behaviours, such as flank attacks, rear attacks and skirmish attacks. These behaviours increase the level of complexity of the battles which also provide more diversity in terms of tactics and battles, making the game more interesting for players. [8, p. 88].

2.1.2 Quadtree

Throughout a battle the battle contingents may collide against each other being necessary to check multiple times the existence of collisions and instead of making a collision detection for each battle contingent in the battlefield, a Quadtree is used which is very efficient for collision detection in a bidimensional space.

The Quadtree divides recursively a bidimensional space into four equal quadrants. In terms of code, a Quadtree is represented by a tree structure where the root of the tree represents the region to divide while the other nodes represents a quadrant [9].
The Quadtree implemented in the simulator has a maximum depth of four which creates a tree with

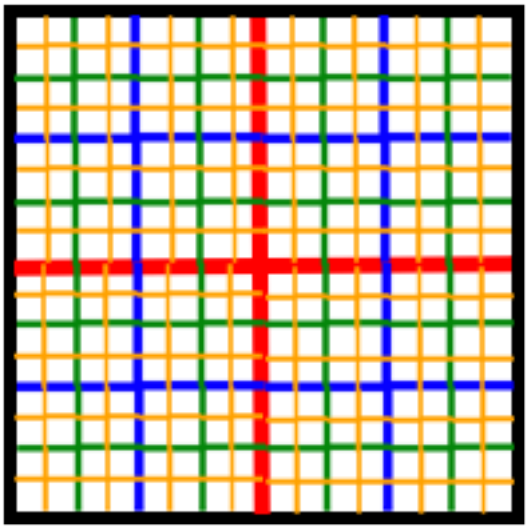

Figure 3: Division of the battlefield by four times.

five node levels and each parent node generates four child nodes where each child node represents a quadrant. At the maximum depth, there are only leaf nodes, which are nodes with no childs, where each node will contain the battle contingents information from a quadrant.

To check if a battle contingent belongs to a certain quadrant, the four vertexes of the rectangle that represents the battle contingent are used and if at least one of them are in the quadrant then the tree will continue expanding until reaching the leaf nodes where the battle contingent information will be kept.

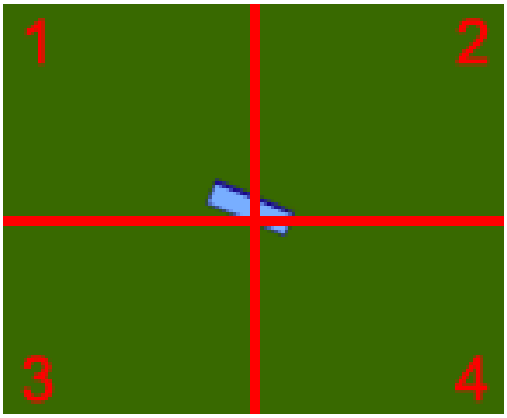

Figure 4: Battle contingent's vertexes placement with quadtree division.

The image above is an example of the situation explained in the previous paragraph where the battle contingent is in three quadrants, two vertexes in quadrant 1 and one vertex in quadrant 2 and 4, which means that its information will be kept in these quadrants.

2.1.3 Game Engine Issues

The issues with the battle simulator were related to performance because when two players were battling each other there was the possibility to exist a high number of contingents and each of these was making their own decisions.

To solve the identified issues, it was decided to reimplement the JavaScript battle simulator in a new programming language, in this particular case the new language was Go, to achieve one of the goals and hoping for an improvement of the simulator's performance.

2.2. Programming Languages Analysis

After an analysis to the game engine it is time to analyze the new language and the previous languange to identify the differences between them and also to verify if it is really worth it to conclude the incomplete version of the battle simulator developed in Go.

2.2.1 JavaScript

The previous language of the battle simulator was JavaScript which is a dynamic programming language, that is, a language that executes certain behaviours, such as adding new code, during the execution time, something that static languages can only do in compilation time. [4].

This language is often used for web browsers because it allows the usage of scripts in the client side to interact with the user, to control the web browser, to communicate asynchronously and to modify the document content that is being displayed. It is also used in the server side through the use of frameworks such as Nodejs, for game development and for desktop and mobile applications [7].

2.2.2 Go

The Go language, also known as Golang, is an open-source language that was developed and announced by Google in November 2009 [6]. This is a statically-typed language which means that the data type is verified in compilation time, which allows the early detection of errors without any

performance costs [10].

The syntax of this language is identical to the C language which makes the learning and adaptation processes easier and because of this the number of developers using Go increased. When the number of developers increase the number of information through the web tends to increase as well which helps the novice developers to give their first steps using this language.

2.2.3 Parallelism *vs* Concurrency

In the battle simulator what would solve the performance issues is to parallelize the current code so it is necessary to give an explanation on how the parallelism works on both languages.

Go has its own scheduler that is responsible to manage the goroutines that are used in parallelism and concurrency.

A goroutine is a lightweight version of a thread and, as it was said before, they are managed by the Go scheduler. The goroutines are executed in the same address space and the access to shared memory must be synchronized, this can be made using sync libraries that comes with the language or using channels.

A channel creates a connection where it is possible to send or receive information using the operand "<-" which is used to synchronize the goroutines without using the traditional locks system.

To add parallelism it is required to say the number of cpus which represents the number of threads that it is going to be used during the runtime by the *GOMAXPROCS* function.

JavaScript has no support for parallelism by itself but nowadays exist libraries for Nodejs that try to make the parallelism in this language a dream come true. The most known library is the Parallel.js which makes use of Web Workers. Web Workers can be used in both web browsers and Nodejs and execute tasks in background.

In general, everytime a Web Worker is created a new thread is also created where it will run some code from a different file and it is also possible to communicate between other Web Workers and the main thread [11].

The Parallel.js tries to reduce the level of complexity of the Web Workers by using high level functions. This library is not available for all the versions of web browsers and it is client-side only which can be considered as a disadvantage.

Unfortunately, there are no libraries for Node.js which offers parallelism with such a simple interface for server-side web applications but there is still possible to get parallelism using other libraries or modules such as cluster.

Since the Node.js operates in a single thread, the cluster module takes advantage of multi-core systems to launch a cluster of Node.js processes. The cluster module supports two methods of distributing incoming connections, the first method is the round-robin approach where the master process listens on a port, accepts new connections and distributes them accross the workers and the second method is where the master process creates the listen socket and sends it to interested workers [2].

3. Implementation

The first step to take was to know the current state of the simulator in Go and this version was compiling without any errors but it was producing invalid battle outputs where most of the battle contingents were not even drawn and they would not execute any of their actions. After knowing the state of the simulator, it was time to create a plan to guarantee that the development went as smoothly as possible.

It was decided that this version must have the same functionality as the JavaScript version so the process consisted in coverting code from JavaScript to Go with some adjustments and improvements during this process.

The duration of the debugging process was too long because the issues were all over the code and to guarantee that every issue was found and solved, all the code was inspected. Having the other version also helped a lot since it was possible to use it to compare results. With this process it was possible to develop a fully functional version of the battle simulator in Go.

The main function of the simulator can be divided in three blocks: preparation & positioning, battle processing and logging battle results.

The first block is responsible for the creation of the battle between players and it initializes all the necessary variables for the battle. Then, it

will sort the slot list for each player to guarantee that the slots with the best rank comes first in the list. Finally, a side is assigned for each player and their army positioned in each side of the battlefield.

To position the troops in the battlefield it is required to make some calculations to put the troops in the right slots and also to calculate their initial position. The first thing to do is to split the troops into categories and for each category it will select the best slots. After this selection, it is time to create the battle contingents and associate each of them to a slot. The creation of the battle contingents starts by counting the number of troops from a category and calculates the maximum number of troops each battle contingent can have and this number varies with the category. The next step is to split the troops into battle contingents and insert them into a list. Then this list will be sorted by Power in a descending order. Finally, the battle contingents can be positioned in the side of the battlefield that was assigned to each player.

After this stage is complete, the block for the battle processing will start and in terms of code is just a loop that executes a battle step, which in other words means that all the battle contingents in the battle will execute their actions one time. At the end of each step, the simulator checks if there is a winner or if the number of maximum steps has been reached in order to end the battle.

Each battle contingent executes four different actions following a defined order:

1. Pick an enemy battle contingent to attack;
2. Move;
3. Attack;
4. Update internal state.

To execute the first function, the battle contingent has to access all its neighbors and then verify which of these are the closest enemies being these inserted into a list of possible targets to attack.

In order to get all the neighbors of a battle contingent, it was used a data structure known as Quadtree and it was previously explained.

When a battle contingent is inserted in the battlefield it is also inserted in the Quadtree and everytime a battle contingent updates its position it also updates its data from the Quadtree and because of this each battle contingent can be removed or inserted multiple times in the Quadtree throughout the battle.

The next action to execute is responsible to make the movement forces calculation which are going to be applied to a battle contingent making them to move. These forces can be rotational forces which cause a battle contingent to rotate, moving forces which make a battle contingent to move and BOIDS-like [3] forces which simulate the cooperation and formation between battle contingents. After this calculations, it is time to try to apply these forces and before doing that it is required to make some collisions verifications. If these forces are applied to a battle contingent and it collides, then the algorithm will try to just apply rotational forces and if the battle contingent is still colliding another battle contingent, then it does not move in this step, otherwise the forces will be applied causing the battle contingent to move.

The BOIDS is an algorithm that simulates the flocking behaviour of the birds and it is a good example of emergence [1]. The emergence is the process where larger entities, patterns and regularities arise through interactions between smaller or simpler entities [5].

In the attack function, the battle contingent will attack all the enemies that are in the weapon range. The damage that a battle contingent can cause to another is modified by equipments, current order and current state and this damage is divided by the number of enemies set as targets. The damage dealt to the enemies may cause casualties.

After every battle contingent execute their actions, all the information of a step is saved in a map and after that it verifies the existence of a winner. When a winner is determined or the maximum number of steps are reached, all the information in the map will be written into a log file which is going to be used later by the Battle Viewer. The Battle Viewer reads the log file data and creates a battle animation showing all the events of the battle to the players.

3.1. Differences between versions

After finishing the development of the battle simulator in Go it was possible to see that there were two major differences between this version and the JavaScript version which were visible in the battle outputs.

The two versions were producing different sorting results for the slot list and different outputs from the cosine function which affected the positioning

from the battle contingents throughout the battle making the battle contingents to have a behaviour variation and also an increase or decrease of the number of steps

The first difference created different initial positioning for the battle contingents, given that the slots of each player were sorted by their rank in a descending order, thus, from the best slot to the worst slot and in the beginning it was possible to have multiple slots with the same rank which led to different sorting results from both languages, despite both languages using the same sorting algorithm. This behaviour proved that even using the same sorting algorithm it can lead to different results because each language has their own optimization of this algorithm which also means that each language deals with multiple repeated elements in a different way.

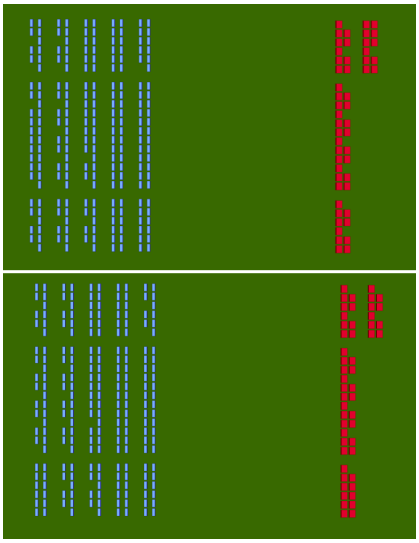

Figure 5: The image on the top shows the positioning of all battle contingents in the battlefield generated by JavaScript while the image on the bottom shows the positioning of all battle contingents in the battlefield generated by Go.

In the figure above, it is visible the differences of the battle contingents positioning caused by the previous issue. Although these differences are not very significant they can create a variation of the number of steps between the two battle simulator versions. Despite the existence of this issue, it is not very alarming since it is not visible to the players because they will never have the opportunity to compare the results from the two versions.

The other difference was related to a problem with the cosine function which also caused positioning changes and a variation of the number of steps. This issue was only visible in long duration battles having no impact in short duration battles.

Example:

Go

```
math.Cos(-0.1) = 0.9950041652780257
```

JS

```
Math.cos(-0.1) = 0.9950041652780258
```

The example above shows the value difference of the cosine function for the same input in both languages. To know if this issue was caused by the battle simulator versions or by the languages itself it was decided to test this same example in an online code playground. Firstly, the Go language was tested and the final result of the cosine for the same input both locally and on playground was the same. The next step was to use the previous method but this time for JavaScript and the final results were different. With this, it was possible to conclude that the cosine function in the JavaScript battle simulator version was returning different results.

This slight difference originated uneven battle results because there were rotational forces that were applied to the battle contingents whose forces used the cosine and sine functions and since each battle simulator version had a different result from cosine it produced different battle contingent positioning throughout the battle.

Figure 6: The image on the left shows the last step of the battle in Go while the image on the right shows the last step of the battle in JavaScript.

The image above shows an example of a 2 versus 1 battle situation and for the same input it is possible to see that in the last step of the battle the positioning of the battle contingents are not the same in both versions because of the cosine issue explained in the previous paragraph.

One of the things that was also changed when developing the new version was the code variables organization through the usage of structures.

4. Parallelism Implementation

After solving most of the issues with the new version in Go, the next step was to insert parallelism in order to get a performance improvement and to make the game more appealing to players, specially the new ones.

In order to insert parallelism it was necessary to create a goroutine, which is a lightweight thread, for each cpu where each of them will be managed by the Go Scheduler and to share information between goroutines it is necessary the use of channels which are used to send values from one goroutine to another.

```
Parallel: P
Single-Threaded: ST

execute_actions() {

    all_update_target(); - P

    all_move();    - ST
    all_combat(); - ST

    all_update_status(); - P
}
```

Since there was dependency problems, the final solution for this problem was to use a mixed solution between single-threaded and parallelism in order to obtain some performance improvements and also to guarantee the correctiveness of the battle results. Probably there would be a better way to solve this issue but, once again, this solution showed good results.

5. Results

After finishing the development of the new version it was time to put it online and use some battle inputs to test it and also to make a comparative analysis between the versions.

During this process there were three versions of the battle simulator tested, one was the original version developed in JavaScript and the other two were developed in Go where one is identical to the JavaScript and the other is a version with parallel programming using four cpus. The test process consisted in using the battle inputs from the game and locally run these inputs between the different versions, registering their execution times.

After running all the tests, a table was created with the registered execution times from each version and with the battle information from each input. With this table it was possible to create graphics to better visualize the difference between the three versions.

Before getting to the tests results, it is important to explain some battle characteristics from each battle input used. These tests were divided into two groups, the controlled battle tests and the random battle tests. In the first group, there are five battle tests where both players use the same tactic and troops and the only thing that changes from battle to battle is the number of battle contingents. In the second group, there are eight battle tests that were picked randomly from the game. For each battle test, there are two images that shows the initial battle contingent positioning and the final battle contingent positioning.

The controlled tests were created to compare the performance of Go and Go-Parallel versions when testing battles with the same characteristics but with different numbers of battle contingents while the random tests were used to compare the performance of JavaScript, Go and Go-Parallel versions with different kinds of battles.

5.1. Controled Battle Tests Results

After running these tests, it was possible to create a graphic showing the execution times for the five battles and also showing the battle duration variation between them regarding the number of battle contingents.

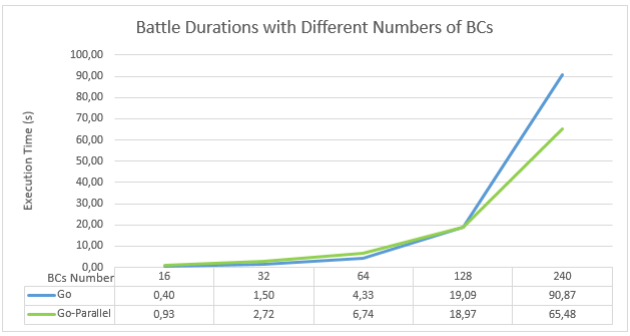

Figure 7: Graphic showing the performance from Go and Go-Parallel versions for the same battle but with different numbers of battle contingents.

The graphic above shows two important aspects, the first aspect is that for the same battle char-

acteristics but with different numbers of battle contingents the execution times from both versions increases and the second aspect is that the Go version was the fastest executing battles containing less than 128 battle contingents while the Go-Parallel was the fastest executing battles containing 128 or more battle contingents. Overall, the Go version was up to 57% faster than the Go-Parallel in battles with less than 128 battle contingents version while the Go-Parallel version was up to 27% faster than Go version in battles with 128 battle contingents or more.

The first aspect showed that increasing the number of battle contingents it will affect directly the battle duration which is expected because when the number of battle contingents is increased the time to process a battle step is also increased and consequently increases the overall battle duration.

The second aspect showed that the Go-Parallel version when dealing with short battles duration and with a short number of battle contingents to be slower than the Go version and this happens because of the overhead added when using with parallelism but, on the other hand, when dealing with long battles durations and large numbers of battle contingents the parallelism pays off.

Another graphic was created but this time using the execution times and the number of steps from each battle to calculate the average time spent from each battle step in Go and Go-Parallel versions.

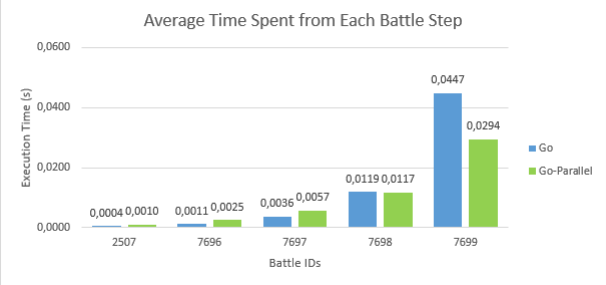

Figure 8: Graphic showing the average time spent, in seconds, from each battle step in Go and Go-Parallel versions using the same battle tests as above.

The graphic above shows the same result as the previous one which was expected because the number of battle contingents has a direct impact in the battle's execution time, in other words, if the number of battle contingents increases, the battle's execution time also increases.

5.2. Random Battle Tests Results

From the selected battle tests it was possible to create two graphics, one for short duration battles and the other for long duration battles, showing the differences of the execution times between the JavaScript, Go, Go-Parallel versions.

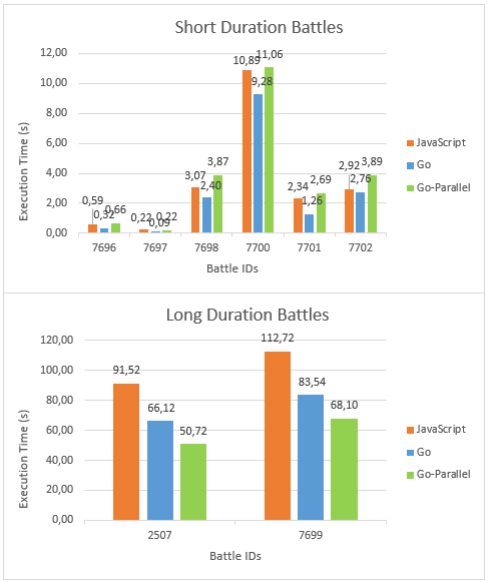

Figure 9: Graphics showing the performance from JavaScript, Go and Go-Parallel versions in short and long duration battles.

In the short duration battles, the graphic shows that the Go version was faster than the other versions showing a performance improvement between some milliseconds, visible in the battle 7697, and a couple of seconds, visible in the battle 7700, while the Go-Parallel version was the slowest version. In the long duration battles, the Go-Parallel version was the fastest version showing a performance improvement greater than fifteen seconds.

In both graphics, when comparing the Go and JavaScript performance it is possible to see that the Go version had the best performance in all the battle tests, with a performance improvement up to 46%. This result shows that the new language offers a better performance than the original language. The Go-Parallel version is slow in short duration battles because of the overhead added when using parallelism and the performance improvements from the parallelism are only visible in long duration battles, with improvements up to

23% when comparing with the Go version.

The next graphic is similar to the graphic from the previous section where it uses the execution times from the battle tests and the number of steps from each battle to calculate the average time spent from each battle step in all the versions of the battle simulator.

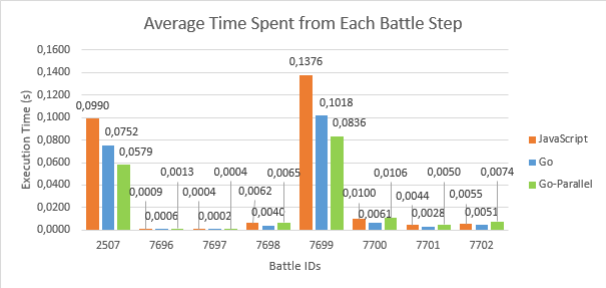

Figure 10: Graphic showing the average time spent, in seconds, from each battle step in JavaScript, Go and Go-Parallel versions using the same battle tests as above.

Comparing this graphic to the previous one, it is possible to see that the results are the same because, as it was said before, the execution time is dependent on the number of battle contingents in a battle.

Having a final look at the results, it is possible to conclude that the Go version has a better performance than the original version in JavaScript in all the tests made. Another conclusion is that the Go version outperforms the Go-Parallel version when executing short duration battles with a small number of battle contingents, on the other hand, the Go-Parallel has a better performance executing long duration battles with a large number of battle contingents.

Despite having battles over one minute even with parallel programming, overall these tests showed a good performance from the new programming language which was one the main goals from this thesis but it also shows that these versions solve some performance issues regarding the JavaScript version, but the Go-Parallel version still needs some improvements in order to execute the battles as fast as possible until it achieves the desirable performance.

6. Conclusions

This document tried to include all the aspects related to the battle simulator module from "My Army", explaining all the decisions made as well as the issues found along the way.

Before starting developing, the first stage was to study the original battle simulator in order to understand how the battle simulation worked and also to detect any kind of issues or improvements. At first, this sounded like an easy and fast task to do, but in reality it took much more time than it was expected because there were a lot of hidden details that it was need to understand beyond the functionality of the simulator, but in the end everything went well.

After understanding all the details and issues related to the simulator it was time to create a solution to achieve the main goals. The final solution was divided into two steps. The first step was to change the programming language from the battle simulator from JavaScript to Go and the second step was to insert parallelism in the Go version in order to get the best performance as possible.

The development process was long and with a lot of obstacles to overtake, but with a lot of effort the Go version was completed. Later on and with some struggle, the Go-Parallel version was created. In the end, the final result was two battle simulator versions, both developed in Go, where one version contains the same functionality as the initial version of the battle simulator developed in JavaScript and another version that uses parallel programming.

With the new developed versions it was time to test them and to create a comparative analysis that helped to take some conclusions. The comparative analysis showed that the Go version had a better performance than the JavaScript in any of the battle tests used which showed that the new language could outperform the original language and it also showed that the parallelism produced worse results in short duration battles than in long duration battles.

Despite the performance improvements obtained from Go and Go-Parallel versions, there is still some work to be done. The main problem with this game is the lack of scalability because as the user keeps playing his army tends to grow and consequently the number of battle contingents increases and with the current performance from the

Go-Parallel is not enough to run battles below the one minute which was one of the goals of this thesis.

After all this process there are still some improvements that could be made to the parallel version of the battle simulator as future work, such as:

- **Move the parallelism to the quadtree:** Everytime a battle contingent updates its position, it is removed from the quadtree and inserted again with the current position and this process takes some time. So a viable solution is to parallelize the current quadtree in order to make the nodes exploration faster;
- **Split the battle contingents list into four quadrants list:** Basically instead of running a single list of battle contingents from each player, the idea behind this is to create four battle contingents lists, one for each quadrant, and then run them in parallel. Theoretically, this should improve the performance of the parallel version;
- **Insert a maximum battle duration or limit the number of troops/battle contingents per battle:** Since this is a real-time game and in case none of the previous suggestions helped, then the only solution is to supress the number of troops/battle contingents in a battle or to insert a maximum battle duration which will avoid all the performance issues.

Finally, it is possible to conclude that most of the main goals of this thesis were fulfilled except for the one stating that battles must execute below one minute.

References

[1] Boids - Wikipedia, the free encyclopedia. Consultado em 15/06/2015. https://en.wikipedia.org/wiki/Boids

[2] Cluster Node.js v4.1.1 Manual & Documentation. Consultado em 28/09/2015. https://nodejs.org/api/cluster.html

[3] Craig W. Reynolds. Flocks, Herds, and Schools: A Distributed Behavioural Model. http://www.red3d.com/cwr/papers/1987/SIGGRAPH87.pdf, 1987

[4] Dynamic programming language. Consultado em 30/12/2014. http://en.wikipedia.org/wiki/Dynamic_programming_language

[5] Emergence - Wikipedia, the free encyclopedia. Consultado em 15/06/2015. http://en.wikipedia.org/wiki/Emergence

[6] Go (programming language). Consultado em 21/11/2014. http://en.wikipedia.org/wiki/Go_(programming_language)

[7] JavaScript (programming language). Consultado em 30/12/2014. http://en.wikipedia.org/wiki/JavaScript

[8] João Morais. My Army: A Game for Social Networks, 2011

[9] Quadtrees and Octtrees. Consultado em 12/06/2015. http://http.cs.berkeley.edu/~demmel/cs267/lecture26/lecture26.html

[10] Static Type system. Consultado em 21/11/2014. http://en.wikipedia.org/wiki/Type_system#Static_type-checking

[11] The Basics of Web Workers - HTML5 Rocks. Consultado em 24/06/2015. http://www.html5rocks.com/en/tutorials/workers/basics/