Game Simulation Engine Optimization

Miguel Cartaxo
Instituto Superior Tcnico, Universidade de Lisboa

Abstract. This document describes the optimization process done on battle simulations of an online multiplayer strategy game set in medieval times, with the main goal of improving significantly the performance of those simulations. The process started with a performance analysis of the current simulation’s algorithm to identify possible problems. To solve these problems two different approaches were taken. With those implemented, another performance analysis was done, with battles scenarios from the game, with positive results.

Keywords: Optimization, Performance, Simulation, Battle, Concurrency, Parallelism, Go

1 Introduction

My Army is an online multiplayer strategy game where the players play the role of a military general controlling an army in medieval times. This type of games, online multiplayer games, can potentially have thousands of players playing at once, which is technically a hard thing to achieve while maintaining a high quality standard where the players’ experience is smooth and responsive. Servers receive many requests in small periods of time, not only for the normal interaction with game, like interacting with the user interface, but also to simulate battles. Strategy games, on the other hand, tend to have many agents represented on the screen at the same time and use several complex algorithms and formulas to make those agents interact in a predictable and real way.

Making all these features work in a game with high performance naturally requires a careful design and implementation where specific tasks should be defined to address the performance qualities of the system.

1.1 Objectives

The main objective is to improve significantly the performance of the battle simulation, desirably by three or four times, through code optimization and introduction of parallelism, while still maintaining the realism of battles. As secondary objective, decreasing the memory usage per battle would also be desirable. To achieve these goals, a set of tests needs to be created in order to measure the performance of battles with different characteristics and it should be possible to compare different optimizations and their impact on performance easily, in order to choose those that yield better results.
2 Related Work

2.1 The Game

The game in question is a browser strategy game, where each player is a military general and controls an army. The player earns gold, the currency of the game, by going into battle with other players and can use that gold to buy more contingents, of various types, equipment and tactics. By winning battles, the player increases rank and climbs to the top the game’s leaderboards.

When the players start a battle, they must choose which contingents will be used in the battle and which will be tactic followed. Both decisions have impact on the battle outcome. After that, the battle is simulated, using a simulation engine, and the outcome calculated and, finally, both players can watch the battle using a replay tool. The players cannot control any troops during the engagement. Both players can also see other statistics, such as, how many kills each contingent had, how many troops it lost and how much experience it got. By winning battles, the players increase their rank and climb to the top the game’s leaderboards.

2.2 The Simulator

A battle in game is simulated using an engine, also referred as the simulator, which originally was written in JavaScript and a different version as implemented by Alexandre Freitas written in Go [7]. The latter is the version that will be worked on. When a battle is finished, it can then be seen through a replay tool for players to see how it all unfolded, Figure 2.2.

![Fig. 1. A battle seen from the replay tool with different types of units.](image-url)
Listing 1.1. High level view of how the engine processes a battle.

```go
func Simulate(player_a, player_b *Player) {
    placesArmy(player_a)
    placesArmy(player_b)

    for every BC {
        update_target()
        move()
        combat()
        update_status()

        if winner == WINNER FOUND{
            break
        }
    }

    populateResults()
}
```

Listing 1.1 shows how a battle is processed, in the current version of the engine written in Go, where after the battle is created, the engine applies the positioning algorithm to both players and then stays in a loop, executing the players’ plans, until a winner is found. When it is, the battle has finished and it outputs the results to text file.

2.3 The Go programming Language

Go is an open source programming language created by Robert Griesemer, Rob Pike and Ken Thompson in 2007, at Google [15]. It was created because these programmers thought that programming had become a difficult and complex task and that the available languages were not considered very good alternatives, since they felt that programmers had to choose between three key features: efficient compilation, efficient execution, or ease of programming; which were not present in one language.

Go is a compiled, statically typed language, which means that types of functions, variables and arguments must be declared and that the types of assigned values and arguments are checked before the program’s execution. It supports networking and multicore computing and aims to be modern, fast and easy to make simple and reliable software.

Go also supports tools for testing and profiling, which are very important because they can prove useful in finding and analyzing possible bottlenecks.

2.4 Feature Toggles

Feature toggles are an important technique to consider in the context of this project because the optimizations will be introduced gradually and several tests
will be done with different combinations of optimizations many times and toggles may help improve the workflow. The basic idea behind feature toggles is to allow the programmer to maintain several “feature branches”, without having to deal with a myriad of versions of the source code, one for each combination necessary for testing. Basically, features are easily turned on or off by toggle points. This can be useful if the programmer wants to only make some features available to a subset of users or testing different versions of the same feature [9] [14].

3 Solution

To achieve this thesis' goals, an analysis was done to know what and where to optimize. For that, it is necessary to have a methodology in order to achieve the best results. The proposed methodology is as follows:

![Fig. 2. Proposed methodology.](image)

*Measure* means finding out what the simulation engine is doing in its current state, e.g. how fast it runs, how much memory it uses or if it outputs the correct results and with that, it is possible to analyze it and identify bottlenecks. Measuring will be done by combining CPU and memory profiling with the use of timers introduced in key places in the code, since the profiles have a function level granularity and it is useful to have a higher level understanding of the cost of a group of functions. As an example, the Update Target is an algorithm step that encompasses several different functions that have the same goal, update a contingent’s target, and it is easier to find out if this step is a bottleneck by introducing timers at the beginning and at the end of this step and see how much time it takes than sift through a big list of functions that are not organized by context, which is what the profiles are. The decision of what to optimize will be based on the results of what was measured and by reading and understanding the engine’s code because there might be an opportunity to make smaller optimizations on portions of code that are not shown as big bottlenecks in the profiles but have unnecessary computations that can be removed. The specific implementation of the optimizations will depend heavily on the context but after it is done the next step is to measure again to know for sure if performance has been improved.

This methodology will be applied to two different approaches of optimization: the first will be to optimize the individual algorithm steps with minimal changes
to the algorithm behavior to avoid having drastically different battle results when comparing to the original version; the second will be a more experimental approach with much less concern for the battle results changes and take more advantage of Go’s concurrency model.

A few battle scenarios were created in order to measure the initial performance of the game’s engine and with that find bottlenecks and identify what needed to be optimized. These scenarios continue to be used during development even though the more important tests would be done with battle scenarios retrieved from the game.

4 Sequential Solution Optimization

The goal of this approach is to optimize the engine with minimal changes to the algorithm behavior. Each optimization implemented corresponds to one feature toggle. In this section we will only go into detail about some of those optimizations.

4.1 Update Target Parallelization

After noticing that the Update Target was taking most of the battle’s time, it was the first portion of the engine that made sense optimize. This step does two things: populates the BC’s list of enemies inside its zone of control and selects an enemy target from a list of possible targets. The most expensive is the latter, since it does a lot of distance calculations, checking if a target is within range, if the BC has line of sight and so on. For the most part, these are independent computations, so introducing concurrency in this step would be fairly easy. The approach was simple, launch a go routine for every battle contingent and wait until every contingent had selected his target before proceeding to the next step, Move.

4.2 Move Forces Parallelization

The initial analysis also showed that the algorithm’s Move was a very expensive step that made sense to focus on and try and make it faster. There are two main reasons for the bottleneck in this step: calculation of boids forces, which involves several distance calculations; and collision detection, both heavily depend on the number of neighbors each BC has at any given time, meaning, the number of other BCs that are in the same quad tree quadrant. A greater number of neighbors means that more time is spent in this step. The approach to optimize was similar to the Update Target parallelization, every contingent does their computations independently and they sync with each other before any of them go to the next step, the Combat.
4.3 Smarter Allocations

This engine makes heavy use of two data structures: maps and slices. Maps, as mentioned before, are a data structure that associates keys to values, and slices are just like arrays but with a more powerful interface and support for more operations, like appending values. Both these data structures can grow in size to accommodate new elements added to them, but if the size value of the maps and slices used is known during the lifetime of the program, an initial capacity can be given to those data structures, basically doing a preallocation, to avoid resizes, more specifically, to avoid calling Go’s runtime functions such as \texttt{growWork}, \texttt{hashGrow} and \texttt{growSlice}. That is what the \texttt{SmartAllocs} toggle is about: going through the code, finding where maps and slices are created and give them an initial capacity value based on what size those structures tend to have during runtime. The expected result here is that by avoiding doing resizes there will be some performance improvements. Now, since the given capacity values are approximate values, there is a chance that in some battles the capacity is never reached and therefore the engine used more memory than it needed or, the capacity is not enough and the maps and slices need to be resized. For the first case, the amount of unused memory will be very small and for the second case the number of resizes will also be very small and so will be the performance impact of those resizes.

4.4 Simplifying Map Iteration

A significant part of the main loop for each BC is getting information from its neighbors, other contingents that are positioned near the BC, for collision detection, boid forces calculation, target choice and so forth. Most of this involves looping over the BC’s neighbors map and one of these instances is in the \texttt{populate\_zoc} function, Listing 1.2.

\begin{lstlisting}[language=golang]
func (bc *BattleContingent) populate\_zoc(battle *Battle) {
    //Non-relevant code
    ... // Relevant code: finding enemies in zoc
    neighbors := battle.keys\_qt(neighbors)
    for i := 0; i < len(neighbors); i++ {
        ct := neighbors[neighbors[i]]
        if (ct.battle\_side != bc.battle\_side &&
            ct.tsm.Current\_state != STATE\_DEAD) {
            if ct.form.fine\_collision(zoc) {
                bc.enemies\_in\_zoc = append(bc.enemies\_in\_zoc, ct)
            }
        }
    }
}
\end{lstlisting}

The \texttt{populate\_zoc} function, which is called in the Update Target step, calls the \texttt{battle.keys\_qt} function which given a map, in this case the BC’s neighbors,
returns a sorted list of the keys of that map, the neighbors’ ids. Then, it loops
over those ids and checks if a contingent, \( ct \), collides with the zone of control,
a small rectangular area around the BC, if so, it adds the contingent to the
\texttt{enemies.in.zoc} slice. This is a wasteful way to deal with the neighbors maps
because the engine needs to iterate through the neighbors more than once, it
needs to allocate a temporary slice to hold the ids and it needs to sort that slice,
unnecessarily so, since it wants to loop through all neighbors, so order does not
matter.

There is no need to call \texttt{battle.keys} because there is no need to go through
neighbors by a specific order or a slice, with the neighbors ids, to iterate through
the map. Using the \texttt{range}, a Go’s statement, in the for loop allows the program
to iterate through the map only once, without the slice and without calling Go’s
runtime sort functions.

These are only some of the optimizations made with this approach with
successful results among many others, but there were some that did not have
a significant impact on performance or changed the outcome of the battle for
worse.

5 Concurrent Design

The second approach used to optimize the game’s engine consists in having
a complete implementation of the game by designing it as a set of concurrent
entities that interact through communication channels. The intention is to verify
if the results of the implementation are consistent with the game rules, and to
measure the performance to compare with the partial parallelization introduced
in the optimized approach. It also presents in detail the problems encountered
and their solutions.

The battle contingents are the only agents in play. This second implementa-
tion will be designed with the concept of each BC running its main loop in its
own go routine. There are three main problems that need to be solve in order
to implement this approach: first, if each BC does its main loop in its own go
routine and therefore asynchronously, how do we define what is a battle step;
second, how is the quad tree maintained up to date, taking into account that
it is shared by the BCs; and third, how do the BCs actually communicate with
each other.

In order to solve the first problem, a \textit{Step Manager} was created, which is a
go routine that receives the BC’s individual battle steps, creating the log, and
keeps track of which BCs have done X number of steps. Note that the BCs that
do more steps in the same amount of time are those that need to perform less
computations, for instance, because they have fewer neighbors. This way, it can
synchronize the BCs in order to avoid situations where some BCs do a lot more
steps than others, making the battle potentially unfair, since this would create
situations where some BCs would do a lot more damage because they played
more steps. The difference in steps between BCs can be define before the battle
starts, but for the purpose of testing the performance of this design, the tests
were done with a difference of one, meaning no BC can do a second step until all the others have finished the previous step. This decision was made because no differences in performance were found when we introduced a bigger difference in steps and, on the other hand, there were problems with the replay tool that is used to view the battle after it is simulated, because it was not prepared for this type of desynchronization between BCs. It was also important that the Step Manager did not write the log to disk, because that takes time and ideally the Step Manager should not be a bottleneck, so a Write Log Process was created, which is another go routine that runs concurrently with the battle, that receives an array of battle steps and writes them to disk.

To solve the second problem, another go routine was created, referred to as Quad Tree Process, with the purpose to receive requests from the BCs, through a channel, every time they move, to update the Quad Tree structure with their new position and send requests to BCs, through another channel, for them to update their neighbors list whenever the changes in the Quad Tree concern them. This way, the problem of multiple BCs writing to the Quad Tree at the same time is resolved.

Solving the third problem is simpler, the way BCs communicate is through channels, so for example, when an attacker deals damage to defender, it sends that damage to the defender’s channel, instead of applying that damage directly to the defender.

It is also important to note that all channels mentioned are buffered, meaning that the go routine that sends a message to a channel does not block waiting for the message to be received on the other end, therefore it continues by doing the next computation. When a message is received it is stored in the buffer and the order of arrival is preserved.

6 Results

The three different implementation, which will refer to as Original, Optimized and Concurrent, were compared, in terms of performance for battles retrieved from the game’s server. These being battles played by real players in a beta version of the game. The purpose of this is to have a significant sample of scenarios to test and therefore more data to reach an informed conclusion about which implementation is better. Forty scenarios were retrieved in total.

Fig. 3 shows the total simulation time of all forty scenarios sorted by the number of BCs. The first thing worth pointing out is that as the number of BCs grows so does the time that it takes for a battle to finish, meaning that the number of BCs in a battle have direct impact on the performance. For scenarios with less than 20 BCs the performance is very similar between all tree implementations, as demonstrated by the high density of markers in that area. As the number of BCs grows both Optimized and Concurrent start to perform a lot better compared to the Original, with the Optimized being 3 to 4 times faster and Concurrent performing 4 to 5 times better.
Fig. 3. Total simulation time Original versus Final versus Concurrent, logarithmic scale.

Fig. 4. Memory Usage Original versus Final versus Concurrent, logarithmic scale.

Fig. 4 shows the memory usage for the same scenarios and engine implementations. It is clear that the number of BCs affects the amount of memory used during battle and that for small scenarios all three versions use, more or less, the same amount of memory. The differences start to become noticeable when the number of BCs in a battle is higher than 20 and get more so as the number grows. On average, the Original, the Optimized and the Concurrent use 11.1MB, 5.7MB and 59.3MB, respectively. The Optimized version benefits from a memory optimization and the Concurrent version, even it has that same optimization, requires more memory due to use of buffered channels.
7 Conclusion

The main goal of this work was to achieve significant performance on an existing multi-agent engine that simulated a battle in medieval times. This document described the approaches taken to achieve that goal.

The results in the previous section showed a noticeable improvement in performance over the original implementation of the game’s engine using both new implementations, although the concurrent design achieved better performance in most cases over the optimized sequential design, at the expense of high memory usage. With these positive results, we can conclude that the methodology worked.

References

1. A Tour of Go. https://tour.golang.org/list