HillClimbing@Cloud

Mafalda Ferreira 81613
Leonardo Epifânio 83496
Pedro Lopes 83540

Group 23

Abstract—HillClimbing@Cloud is an elastic cluster of web servers in Amazon's EC2 Instances that can find the highest point in a given map by executing a set of search/exploration algorithms. It's a system with high scalability, good performance and efficiency, as it performs the selection of the cluster node for each incoming request and manages the number of active nodes in the cluster to have high availability using the least resources possible.

1 Introduction

The goal of this project is to design and develop **HillClimbing@Cloud** - an elastic cluster of web servers capable of finding the highest point on simplified height-maps, using three different strategies: Depth First Search, Breadth First Search and A*. This system will be run within the Amazon Web Services ecosystem, where each ***Web Server*** corresponds to an AWS Elastic Compute Cloud (EC2) instance that will receive web requests from users.

Each request requires a set of information, necessary to draw the search path leading to the highest point on the height-map, represented in Table 1. The corners refer to the active search rectangle within the height-map and are optional, since they're only needed if the user wants to restrict the search to a sub-rectangle.

Parameter	Description
map	The height-map to analyze.
Width	Map total width
Height	Map total width
(x_S, y_S)	Start position
(x_0, y_0)	Top Left Corner [Optional]
(x_1, y_1)	Bottom Right Corner [Optional]
strategy	Desired strategy (BFS, DFS, A*)

Table 1: Necessary parameters to the requests sent to the Web Server.

Each ***Web Server*** is Java-based and will receive the HTTP requests asking to find the highest point, process them and return the result to the clients.

In order to achieve high scalability, good performance and efficiency, the system has a ***Load Balancer*** and ***Auto-Scaling*** mechanisms to optimize the selection of the cluster node for each incoming request and to optimize the number of active nodes in the cluster.

We begin in section 2.1 by describing the distributed architecture of the solution, including the overall control, a brief explanation of the main components and the correspondent data flow. In Section 2.2, is presented a description of the selected metrics used to estimate the requests complexity, including their purpose, conducted tests used to determine which tools to use and the adopted solution. Section 2.3 gives a description of the data structures used to store the instrumentation metrics and the instances objects. In Section 2.4 it is provided a description of the algorithms and heuristics used to estimate the cost of execution of a given request. Section 2.5 explains in detail the developed Load Balancing algorithm and Section 2.6 details the developed Auto-Scaling algorithm. In Section 2.7 we'll describe the implemented mechanisms used to ensure transparency to the clients while the system recovers from incomplete requests due to fails or scaling decisions.

2 System Overview

2.1 Architecture

The system is composed by an arbitrary number of ***WebServers*** and one ***Load Balancer Server***. As we can see in figure 1.

The ***Load Balancer Server*** represents the only entry point into the system. It receives the requests from clients and is mainly responsible for distributing the workload among the ***WebServers***, picking the best node to complete the request (Load Balancing) and is also responsible for deciding how many ***WebServers*** nodes should be active at a given moment, scaling out and in the size of the system, according to a set of rules (Auto-Scaling). This last task runs on a separate thread from the load balance. In short, the ***Load Balancer Server*** performs the load balancing and the auto-scaling.

The ***WebServer*** receives requests from the ***Load Balancer Server***, computes the results, saves the

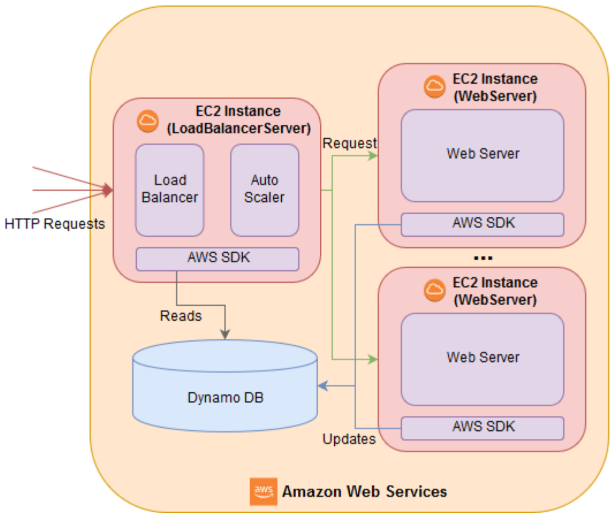

Figure 1: System architecture.

measured *work* in the database, and returns the results to the ***Load Balancer Server***.

2.2 Instrumentation Metrics

To decide the metrics for estimation of request complexity, 9 tests were run for the same map (RANDOM_HILL_1024x1024_2019-03-08_17-00-23.dat). These tested the 3 strategies, for 3 different search area sizes, with the starting point at the middle of the search area:

1) Strategy: BFS, Width: 1024x1024, Starting Point: (512, 512).
2) Strategy: DFS, Width: 1024x1024, Starting Point: (512, 512).
3) Strategy: ASTAR, Width: 1024x1024, Starting Point: (512, 512).
4) Strategy: BFS, Width: 512x512, Starting Point: (256, 256)
5) Strategy: DFS, Width: 512x512, Starting Point: (256, 256).
6) Strategy: ASTAR, Width: 512x512, Starting Point: (256, 256).
7) Strategy: BFS, Width: 256x256, Starting Point: (128, 128).
8) Strategy: DFS, Width: 256x256, Starting Point: (128, 128).
9) Strategy: ASTAR, Width: 256x256, Starting Point: (128, 128).

The purpose of these metrics is to estimate the CPU effort necessary for each request to complete. Because of this, we want the metrics to be proportionate to the CPU time it takes to complete each request. The name we give to the metrics in the context of this problem is *work*. The bigger the accuracy and the smaller the overhead, the better the metric.

The first test we did measured the CPU time for each of the requests (Table 2). Having these values

Request	Time (s)	ICountParallel
1	24.111	440766210
2	7.743	127938277
3	29.414	500178823
4	7.523	123842869
5	10.539	164279699
6	7.987	125077962
7	2.561	30464053
8	3.273	39377427
9	2.964	31309514

Table 2: Time (in seconds) and number of instructions by request, using ICountParallel.

as reference (obtained on a personal computer), we run several instrumentation tools on the solver classes:

General Instruction Count - The first tool we tested, **RoutineICount**, counted the instructions of every routine by accessing the routine's instruction counter. This tool proved to be accurate, despite overestimating the *work* in bigger search area sizes, and presented an overhead of just 1.04. At a first look, this instruction counter may not be the most accurate, since it doesn't account for execution cycles inside the routines. Because of this, we tested an instruction counter that incremented using the size of each block, **BBICount**. To our surprise, the accuracy of this tool was very much the same as the first. This can be explained by the fact that throughout the execution of the cycles in the solver classes, there are various calls to routines, whose instructions are counted each time they are called. Since this last tool had a bigger overhead (1.18), and since counting each instruction at a time had a massive overhead, the best **General Instruction Count** algorithm we could find was **RoutineICount**.

Specific Instruction Count - We developed several tools, each which counted one of the types of instructions defined in the *highBIT/InstructionTable* class. These tools presented a bigger overhead than **RoutineICount**, but no improvements in accuracy, so they were discarded.

Memory Analysis - Measuring Load/Storage info, the most accurate results were achieved by calculating the average of *Field Load*, *Field Store*, *Regular Load* and *Regular Store*. These measurements were not more accurate than **RoutineICount**, and even presented a bigger overhead (2.5), so they were discarded. Counting the number of allocation instructions was also reproved due to its results being contradictory to the CPU Time measurements.

Program Execution Info - Counting the number of executed methods and basic blocks proved to be contradictory to the CPU Time measurements, so those tools were out of question. The number of *taken branches*, however, was accurate, but since it presented a bigger overhead (4.2), it was discarded.

In the end we decided to go with the **RoutineICount** tool, which was then parallelized, changing the name to **ICountParallel**. This tool is only run on the classes *Solver*, *SolverArgumentParser$SolverParameters* and *SolverArgumentParser*. We picked these because they provided good accuracy, and most of the methods being called throughout the execution are located in them. The results of this tool are present in Table 2.

2.3 Data Structures

2.3.1 HcRequest

To save the instrumentation metrics, we used a table in the database system DynamoDB. To save the metrics, an **HcRequest** data object is used. This data object contains all the parameters (Table 1) of a request, plus a **requestId**, which is a concatenation of these parameters, the measured metrics for that request, and a boolean which tells if the request has been **completed**. These objects are only modified on the WebServer, but accessed in both the WebServer and the Load Balancer Server. When a WebServer receives a request, it first checks if the corresponding **HcRequest** object is created in the DB, creating it if missing. If the request has not been completed, the instrumentation tool performs the counting, saving the result to the DB in the end. Otherwise, it does neither of these things.

2.3.2 InstanceInfo

A dictionary mapping instance IDs to these objects, **runningInstanceInfos**, is present in the Load Balancer Server, in order to save, for each running instance, the instance object, the number of currently running requests sent to that object at the time, **numCurrentRequests**, two booleans, **willTerminate** and **isFresh** (which will both be explained in Section 2.6), the last CPU usage measured in that instance, **lastCpuMeasured**, and the amount of current *work* sent to that instance, **work**. When a request is complete, the **InstanceInfo** object of the instance to which it was sent, has its **numCurrentRequests** and **work** decremented.

2.4 Request Cost Estimation

To estimate the cost of a request, firstly, the Load Balancer checks if the request was completed at a point in time, by querying the DB. If it's found, it uses that value. If not, it tries to get from the DB any requests with the same map and similar parameters (x_0, x_1, y_0, y_1, x_S, y_S), by testing if each deviates by $\frac{1}{4}$ of the side of the search area (in their respective axis), of the current request. If it finds any, so called, similar requests, it tries to perform operation **estimateSS** on those requests, if it doesn't find any, it gets the requests with similar size and performs the same operation on these requests.

In case, no requests with similar size are found, it gets all requests with same strategy and performs the **calculateAv** operation on those requests, and in case no requests with same strategy are found, it just performs **calculateAv** on all requests in the DB.

If there are no requests, it uses a default value of work, 200000000, obtained by averaging results obtained in the development of the project.

estimateSS - Get, from the received requests, the ones with the same strategy and perform **calculateAv** of those requests, if no requests with the same strategy are found, it performs **calculateAv** with the original requests.

calculateAv - With the given requests, calculates the average work and approximate average search area size. Then, from these values, a proportionate estimate is calculated. This calculation is represented bellow, being AW the average work of the requests, RS, the current request's size, and AS, the average request size:

$$AW = AW \times (\frac{RS}{AS})^2$$

2.5 Load Balancing Algorithm

The Load Balancer is an HTTP handler that, concurrently, handles requests from clients, by invoking the **handle()** method. When a request is received, it calculates the estimated work for that request, and then tries to send the request to the running instance with the least *work*. If it doesn't find any *valid* instance, it launches one and awaits for it to start. In this waiting process, if a running instance becomes available (*valid*), the request is then sent to that instance.

An instance is considered *valid* if it's running and is not *marked to terminate*.

A request is only sent to an instance with already running requests, if the sum of its estimated *work* and the *work* being performed by the instance, doesn't exceed $MAX\_WORKLOAD$ (whose value is presented bellow). In the opposite case, a new instance is launched. This is done because, in case a request with a very large amount of *work* arrives, we don't want it delaying the requests at other instances, so for that, we reserve a single instance for it.

$MAX\_WORKLOAD$ has a value of 500000000. This value was achieved by computing a BFS on the total area of map RANDOM_HILL_1024x1024_2019-03-08_16-59-31.dat, starting in the opposite corner of the maximum height, covering, possibly, the longest distance possible in the height-maps that we were given, taking a total of 6 minutes and 32 seconds to compute, and calculating a *work* of 498081995.

Throughout this process, when the Load Balancer requests the instance with the least *work*, it updates the map of **InstaceInfos**.

2.5.1 Updated Work Optimization

The concept of this optimization is for the WebServers to regularly update the *work* they have performed for each request, while they compute them. Before the Load Balancer sends a request to an instance, it updates a database object, **HcInstance**, containing the *work* currently being performed in the corresponding instance, by adding the *work* of the current request into it.

In the WebServer side, at each 12000000 instructions of a request, which represent, around, 10 seconds of work if a single thread is running, the WebServer will update the database object of the corresponding instance, by subtracting the batch of *work* performed since the last update. Since there will be several threads performing operations on these database objects, optimistic locking would have to be used.

This concept was implemented, but, due to bugs and shortage of time to solve them, the optimization was not included in the final version.

2.6 Auto-Scaling Algorithm

The Auto-Scaler runs in a different thread from the Load Balancer. This thread contains a cycle which runs, in intervals of 1 minute, the Auto-Scaling routine. Firstly, it updates the InstanceInfos and gets the CPU usage of the last data point of each instance. If an instance has exactly 0 CPU usage, it is marked as *fresh* (boolean **isFresh**, presented in section 2.3.2).

It then runs **checkMarkedInstances**. This method is run to check which instances are *marked to terminate*, and, if there are no requests currently running in that instance, that instance is terminated asynchronously. An instance is *marked to terminate* when the Auto-Scaling algorithm decides that it should be terminated. But, instead of terminating the instance right away (as it can have requests running in it, at the moment), the instance is *marked to terminate* (boolean **willTerminate**, from section 2.3.2). From this moment on, no more requests will be sent to that instance, and the Auto-Scaler will wait for the requests already sent to it to finish, before terminating it.

After this, the Auto-Scaling routine gets all instances that are not *marked to terminate*, and calculates the average CPU usage and work of these instances. If one of these entries is *fresh*, then the algorithm returns here, if else, it checks if the average *work* is bigger than 70% of $MAX\_WORKLOAD$, launching an instance in the positive case. If the average *work* is lower than 30% of the $MAX\_WORKLOAD$, in case there are more than 1 instances running, it marks the instance to terminate, in case there is only 1 instance running, it will only mark this last instance to terminate, if the CPU usage is bellow 50%. This is done to keep the last instance running even while it's only performing a small amount of *work*.

As was stated, the Auto-Scaling algorithm does not use the heuristic CPU usage for the majority of cases. CPU usage is not the best heuristic for this system, due to the fact that even for a small request, the CPU usage in an instance will go up to almost 100%, not being able to make a distinction between instances running a small request, and instances running a big request, or several requests.

These stated values for the upper and lower limits of workload were chosen because they help scale the algorithm before the instances become overloaded. Of course, the best scenario possible, would be to have each instance run only one request at a time, but this would be a waste of resources. In a real world scenario, the monetary cost of having instances running would come into play, and patterns of access and their relation to the complexity of requests, would have to be taken into account to decide how to better scale the system, and adjust these limits, along with the minimum and maximum number of total running instances.

2.7 Fault Tolerance

When a request is sent to a WebServer instance, if the instance, for some reason, stops running (due to crash or shutdown), the Load Balancer holds onto the request and tries again, repeating the process by getting the new least used instance and sending the request to it. This guarantees that, if the Load Balancer doesn't crash, the request will eventually be fulfilled.

3 Conclusion

We presented the design and solution for an elastic cluster of web servers capable of finding the highest point in a map using three time consuming search/exploration algorithms. Despite not implementing the Updated Work Optimization, the features which were asked for in the statement, were implemented, and the system performs well, and without failing. We were satisfied with the results.