Sudoku@Cloud

Cloud Computing and Virtualization Project - 2019-20

Group 16

Instituto Superior Técnico

Lucas Lobo Fell
86464
lucas.fell@tecnico.ulisboa.pt

Pedro Antunes
86493
pedro.c.antunes@tecnico.ulisboa.pt

Rúben Silva
96999
rubenagsilva@tecnico.ulisboa.pt

1 Architecture

The Sudoku@Cloud system will be run within the Amazon Web Services ecosystem. The system will be organized in four main components:

1.1 Components

1.1.1 Web Servers

The web servers receive web requests from the load balancer to perform puzzle solving, discovering missing elements in the grid and return the solved puzzle. After solving a puzzle, the web server stores the collected instrumentation metrics in the metrics storage system (MSS). This information will be useful when the load balancer is estimating the cost of an incoming request.

In Sudoku@Cloud, there is a varying number of identical web servers, a number that is dynamically updated by the auto-scaler.

The web servers run on rented AWS Elastic Compute Clouds (EC2) instances.

1.1.2 Load Balancer

The load balancer is the only entry point into the Sudoku@Cloud system. It receives all web requests, and for each one, it selects an active web server to serve the request and forwards it to that server. Later, when the selected web server finishes solving the puzzle, the load balancer receives the solution and sends the solved puzzle to the client.

Since this is its responsibility, to send the solved puzzle to the client, the load balancer can provide fault-tolerance capabilities in case one of the selected web servers fails while it is executing a request, as we will talk further ahead, in section 7.

For this final delivery, we designed a more advanced load balancer than the one available at Amazon AWS. Our load balancer runs on a shared instance with the auto-scaler and uses metrics data obtained in earlier requests, stored in the metrics storage system, to pick the best web server node to handle a request.

1.1.3 Auto-Scaler

The auto-scaler is in charge of collecting system performance metrics and, based on them, adjusting the number of active web servers to which the load balancer can send requests. It detects that the web app is overloaded and starts new instances and, conversely, reduces the number of nodes when the load decreases.

We believe we implemented a solution that offers the best balance between performance and cost, described in section 6.

1.1.4 Metrics Storage System

The metrics storage system uses one of the available data storage mechanisms at AWS to store web server performance metrics relating to requests, Amazon DynamoDB.

These nodes process the Sudoku@Cloud requests using code that was previously instrumented, in order to collect relevant dynamic performance metrics regarding the application code executed, as discussed in section 3.

1.2 Control and data flow

The overall control and data flow of the solution can be described as follows:

First, the load balancer receives an HTTP POST request from the client containing the necessary information, i.e., the puzzle template (a given sudoku map of a given size), the solver strategy (BFS - Brute-Force Solver, DLX - Dancing Links, CP - Constraint Programming) to use, and the top threshold for the position of the last missing entry, driven by a series of prime numbers (with the default being the total number of entries of the puzzle).

Following the reception, the load balancer estimates the cost that the incoming request will have, taking into account a series of heuristics and using the information present in the MSS, as described in section 4.

After estimating the cost, the load balancer is prepared to choose, from the available web servers, the best one to handle the incoming request. The load balancer knows which web servers are active by accessing a data structure maintained by the auto-scaler with such information.

For this selection, the load balancer considers how many and what requests the active web servers are currently handling, their current progress and, conversely, how much work is left taking into account a cost estimate that was calculated when the request arrived.

With the web server responsible for solving the puzzle selected, the load balancer sends an HTTP POST request to that web server containing the necessary information, previously mentioned.

The web server receives the HTTP POST request and performs the puzzle solving algorithm, specified in the received parameters.

With the puzzle solved, the web server stores the collected instrumentation metrics in the MSS for later use.

After storing the metrics, the web server responds to the HTTP request with the solved puzzle.

The load balancer receives this response to its HTTP request and sends the returned solution of the puzzle to the client, finally answering its HTTP request.

2 Instrumentation Metrics

Time is not a reliable metric to measure load in the cloud. This is due to several factors such as shared systems and frequent resource overcommits, all of which incur VM slowdowns. These result in wall-clock time delays, which naturally influence the processing time of the requests. To address this, we had to find reliable and equivalent metrics that could describe the load independently from the current running variables.

In our approach, we treated the solver code as a black box and instrumented everything together, looking just at the results for the analysis.

The first step to good data analysis is a reliable dataset. To generate our dataset, we ran the solver for multiple combinations of the possible input variables on the sample website:

- Puzzle (9x9_101, 9x9_102, ...);
- Algorithm (BFS, DLX, CP);
- The number of unassigned positions (full, half, and a quarter).

Since we had to match time with metrics, time was measured independently from the rest using non-instrumented code. The studied metrics were the number of methods, basic blocks, instructions, field load, field store, regular load, regular store, new, newarray, anewarray, and multianewarray.

The first thing was to manually analyze the data to see which metrics were potentially interesting or which ones should be discarded right away. Both anewarray and multinewarray were deemed irrelevant for being very close to zero across all entries. All others were considered potentially interesting.

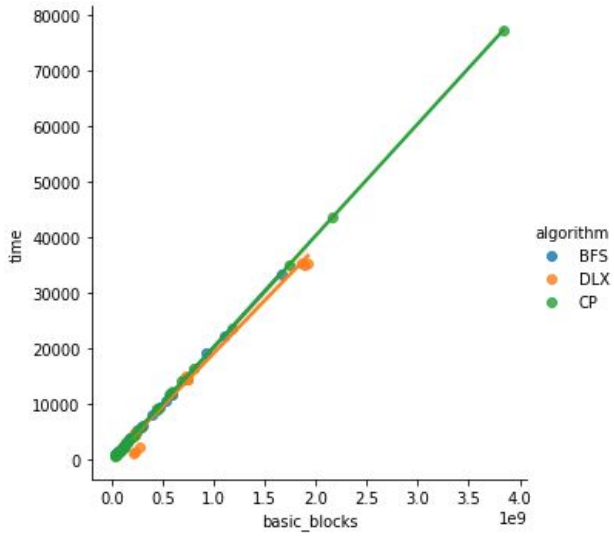

Figure 1: **How the number of basic blocks affects time.**

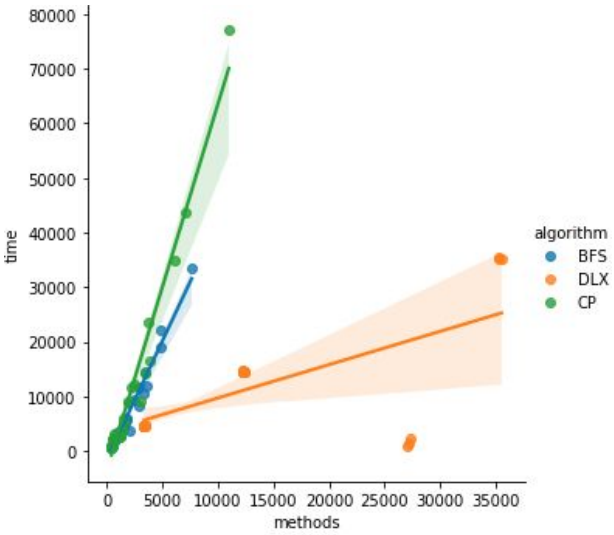

Figure 2: **How the number of methods affects time.**

Locally, time is a reliable way to measure complexity as the other mentioned variables, such as current resource use, can be controlled. For this reason, we tried to find a correlation between time and all metrics. This was done separately for each solver algorithm. The metric which had the best correlation and was similar in all algorithms was the number of basic blocks and can be seen in Figure 1. Instructions also had a good correlation, but the coefficient

was not the same across algorithms. The problem with these two metrics is that they impose a significant overhead. Since one new instruction is placed next to every instruction, the executing time effectively doubles. After testing, we concluded that the overhead was around 90%. The only other metric which can be used in all algorithms and does not have this overhead is the number of methods. After testing, we concluded that the overhead was around 1 to 2%.

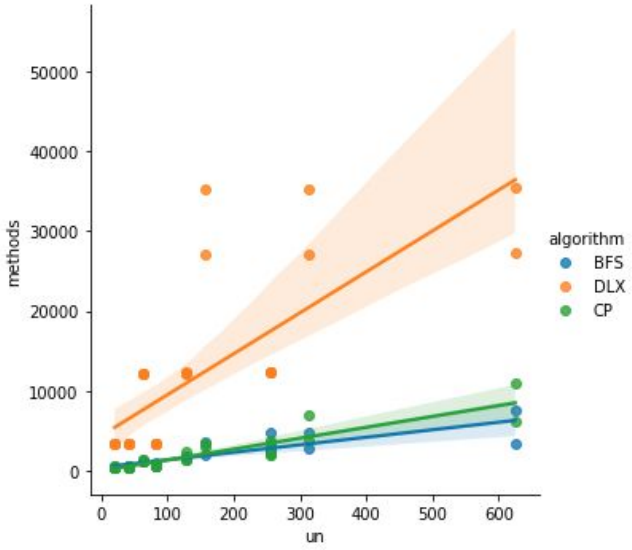

Figure 3: **How the number of unassigned entries affects the number of methods for different algorithms.**

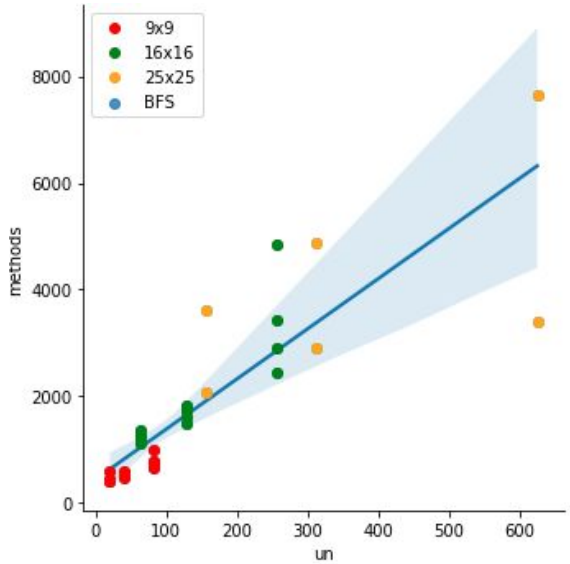

Figure 4: **How the number of unassigned entries affect the number of methods with different sized puzzles in the BFS algorithm.**

The only caveat with the number of methods is that each method has a different cost depending on the algorithm, as seen in Figure 2, meaning a direct cost comparison cannot be made, and that a second transformation needs to be made. First, we need to estimate the number of methods, and then we need to estimate the time based on the estimated number of methods.

To estimate the number of methods based on the input parameters, we plotted the number of unassigned blocks to the number of methods in each respective combination and found that it was generally a good approximation for BFS and CP. For DLX this was not the case. This can be seen in Figure 3.

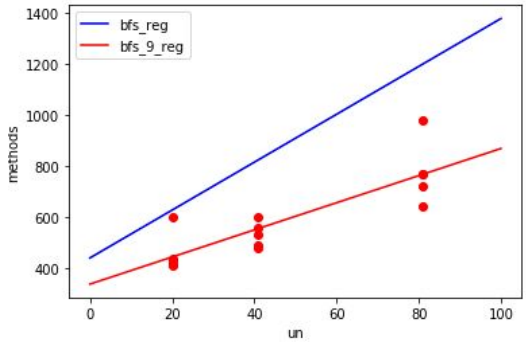

Figure 5: **How the filtered regression differs from the general regression.**

For BFS and CP we created an optional filter for the puzzle size which changes the predictions and increases the accuracy of the model. As we can see in Figure 4, the number of unassigned entries is generally good at predicting the number of methods yet under performs for specific sizes. This is where the filter is a good option. Figure 5 shows the difference between BFS general regression and BFS filtered regression. For this specific scenario, the R2 score of the general regression to the filtered data points is obviously bad, scoring -320%. The filtered regression is much better at 72.52%. This does not mean that the general regression is not good for estimating a random point that might be in the 9x9 size, yet it is prefered to use the most appropriate and existing regression.

For DLX we realized that the only factor which mattered is the size of the puzzle which means that a simple regression was used, as seen from figure 6.

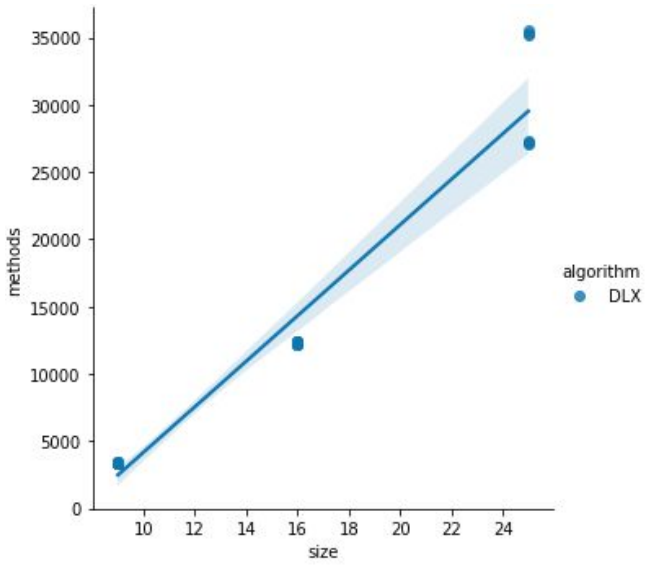

Figure 6: **How the puzzle size affects the number of methods in the DLX algorithm.**

3 Data Structures

The final data structure used was a single table in DynamoDB. This table consists of items, where each item stores:

- incremental request ID
- solver algorithm
- size of the puzzle
- number of unassigned positions
- resulting number of methods

Every time a solver instance completes a request, it sends a new item to the table. The table is fetched periodically to update the regressions mentioned in the previous section. More details on the updating processing are given in the next section.

4 Request Cost Estimation

Based on the insight obtained from analyzing the dataset in section 2, we developed a system of regressions that are used to estimate the cost of incoming requests.

The choice of the regression for each request is dependent on its parameters and the amount of data the system has available at the given time. If the system does not have sufficient data, the most appropriate heuristic is used, otherwise, a continuously updated regression is used.

There are heuristic regressions for BFS, BFS-9x9, BFS-16x16, BFS-25x25, CP, CP-9x9, CP-16x16, CP-25x25, and DLX.

Assuming that the system has no previous data, the logic for the heuristic choice is the following:

- If the request is of the algorithm BFS or CP, and the puzzle size is either 9x9, 16x16, or 25x25, then a filtered heuristic regression (as explained in section 2) for the specific case is used. On the other hand, if the puzzle size is not one of those, the general heuristic regression for the algorithm is used.
- If the request is of the algorithm DLX, then only one heuristic regression exists and that one is used directly.

The continuously updated regressions start being used instead of the heuristic counterpart as soon as a minimum number of samples for that given case are acquired and the variance in the number of methods is higher than 0 (if all samples represent the same number of methods then the regression is impossible to calculate). For DLX, the minimum number of samples is 5. For filtered regression of BFS or CP (like BFS-9x9) the minimum number of samples is also 5. For the general level of BFS or CP (where the size of the puzzle is irrelevant), the minimum number of samples is 10.

A few illustrative examples:

- The request BFS,9x9,50 arrives and no data exists. The filtered heuristic for BFS-9x9 is used.
- The same request arrives and at least 5 samples for BFS 9x9 with a non-zero variation in the number of methods exist. The filtered and calculated regression for BFS-9x9 is used.
- The request BFS,36x36,1000 arrives and no data exists. The general heuristic for BFS is used.
- The same request arrives and at least 5 samples for BFS 36x36 with a non-zero variation in the number of methods exist. The filtered and calculated regression for BFS-36x36 is used.

The continuously updated regressions use data from the previous requests. At the end of each request the MSS is updated, and every 30 seconds the system queries the MSS for updates that were still not propagated to the respective regressions on a background thread. When a point is fetched, its value is used to update both the general regression and the filtered regression, in the case of BFS or CP.

It should be noted that the heuristic regressions are final and are never updated. After a small number of requests the heuristic regressions will most probably never be used again, and are only there because the MSS is not populated from start.

Each regression returns the number of estimated methods for the specified combination of parameters. The regressions are grouped in their respective algorithms estimators, which then convert the number of estimated methods to the estimated cost, as explained in section 2.

This means that after the regression estimates a certain number of methods, this number is converted to a cost that is equivalent between all algorithms (BFS, DLX, CP).

Since a running linear regression was implemented, the cost of updating it is not significant. What this means is that the regression does not need to be computed for every new point, but instead an update occurs. Adding one point to the regression has a complexity of O(1) and recalculating its new parameters also has a complexity of O(1). Adding a set of new points has a complexity of O(N) where N is the size of the set. This is the minimum possible complexity since all points need to be considered. Since the regressions are always ready to use, estimating the cost of a request has a complexity of O(1).

5 Task Scheduling Algorithm

After estimating the incoming request cost, the load balancer must choose, from the available web servers, the most suited to handle the request.

For this selection to be made, the load balancer starts by accessing a set, maintained and updated by the auto-scaler, containing the active worker instances: an active worker instance must be in a running state and must have the image identifier of the image used to create the worker instances.

The task scheduling algorithm starts by traversing this set of active instances, in order to gauge the progress in each instance.

To make this selection, the load balancer accesses the value of the instrumentation metrics, at the given moment, in each of the requests associated with each instance in the set. These values are maintained and updated continually by each worker instance for its requests and this information is sent regularly to the load balancer.

For each request of a given instance, the algorithm subtracts the progress done so far to the initially estimated cost of that request, calculated as described in section 4. These differences, of all the requests of a given instance, are then added up and represent the work left to do for that instance.

Since each worker instance reports its progress using the number of methods counted so far, and the methods have different costs depending on which solving algorithm they refer to, the load balancer needs to convert the number of methods received to its real cost for the comparisons to be valid across requests with different solving algorithms, as explained in the previous section.

Finally, these sums are compared and the instance with less work still left to do is selected for receiving the incoming request. Therefore, the task scheduling algorithm considers how many and what requests the active web servers are currently handling, their current progress and, conversely, how much work is left taking into account a cost estimate that was calculated when the request arrived.

In case one of the active web servers fails while it is executing a request, the load balancer must choose a new appropriate web server to handle that request. When this happens, the task scheduling algorithm helps finding the new worker instance for that request, as explained in detail in section 7.

This way, a worker instance with a lot of work left to do will not be receiving incoming requests from the load balancer. On the other hand, a worker instance that is close to handle all of its requests or that that has no requests associated, will be selected by the load balancer to handle the incoming requests. The distribution of requests across the active web servers will optimise the response time for each request (reducing latency), avoiding unevenly overloading compute nodes while other compute nodes are left idle providing an overall increase in throughput.

6 Auto-Scaling Algorithm

The auto-scaler is running on the same instance as the load balancer and is started when the load balancer starts. From this moment, it gathers all the running solver instances and stores their references in the *readyInstances* map. Two more maps are created, *startingInstances* and *instancesToShutDown*, initially empty.

StartingInstances stores instances which are in their grace period and should not be receiving any requests from the load balancer. *InstancesToShutDown* stores instances which are signaled to be shut down by the auto-scaler but might still have ongoing requests and thus are not safe to be shut down just yet. Once the requests are completed, the instances are shut down and removed from the map.

The auto-scaler enters an infinite loop where one iteration happens every TIME_INTERVAL seconds (which defaults to 10) and during each iteration it will start new instances if the current number of running solver instances is lower than the minimum amount, defined by MIN_INSTANCE_COUNT (which defaults to 1).

Afterwards, for every solver instance that is ready to receive requests it will check its average CPU usage gathered in the three most recent data points and compare them with the minimum and maximum CPU values. If the instance does not have three data points it will try to either use only two or one, depending on how many it has. If the average CPU usage of all data points gathered is lower than the minimum defined by MIN_CPU_VALUE (which defaults to 20), then this instance will potentially be ordered to shut down. If these values are higher than the maximum defined by MAX_CPU_VALUE (which defaults to 60), another solver instance gets initialized if it does not surpass the maximum amount, defined by MAX_INSTANCE_COUNT (which defaults to 10). In the case that one instance is found to be outside the optimal range of CPU (i.e. one instance has to shut down or a new one has to be created), the according action is performed and the remaining instances

are not checked. This means that during each iteration the number of instances changes at maximum one.

An instance only gets shut down if there exists more than the minimum required number of ready instances running. To shut down an instance it waits for that instance to finish all its requests before terminating it, by moving it from *readyInstances* to *instancesToShutDown*. An instance on the *instancesToShutDown* map will not receive any more requests from the load balancer. Every time a request starts or finishes, the load balancer informs the auto-scaler about the responsible instance. This information is used at the end of each iteration to verify whether the instances in the *instancesToShutDown* map are safe to shut down. In the case they have pending requests, the instance is kept alive for at least another iteration.

At the end of each iteration, the instances in the *startingInstances* are verified for their remaining grace period. Once the grace period is over, they are moved to the *readyInstances* map. The grace period is defined by GRACE_PERIOD (defaults to 30 seconds).

When the load balancer determines that an instance has crashed it alerts the auto-scaler by calling the *reportDead* function. The auto-scaler immediately removes from the *readyInstances* variable and drops the instance just in case.

7 Fault Tolerance

When the load balancer assigns a request to a worker instance the query associated with the request is stored in a list held by an object called *InstanceRequest*. This object contains an instance identifier, the queries associated with the instance requests, and requests themselves. In the case a worker instance fails while solving requests, the load balancer can retrieve all the information regarding that request and resend it to another worker instance. If the request is effectively handled, this request is deleted from the list.

When launching the load balancer, a new thread is created and executed. This thread consists of sending health checks to all active worker instances received from the auto-scaler. If a worker instance does not respond to a health check, it will try two more times before discarding it. When discarded, the auto-scaler is warned that the worker instance has failed, and all its requests will be sent to the load balancer for further handling. The load balancer will then repeat the process of choosing the best instance to handle those failed requests.

Finally, this thread will sleep for a HEALTH_CHECK_TIME_INTERVAL which default is 30 seconds before repeating all of this again.