Computação em Nuvem e Virtualização

MEIC-T

Grupo 14

Rafael Soares
87675

Miguel Barros
87691

Gonçalo Pires
94112

Architecture

In this phase, we have the Solver classes instrumented and the WebServer code up and running on the EC2 AWS with the default Load-Balancer and Auto-Scaler.
In the next section we will explain what classes we instrumented, why we instrumented those classes, which metrics we chose and why and how we modified the WebServer to handle multi-thread. Finally, we explain our plans for future implementation.

Instrumentation

We have decided to only instrument the three solver classes: SudokuSolverDLX, SudokuSolverCP and SudokuSolverBFS.
This is due to these classes having the majority of the changes between solver strategies, with the other classes used being common between strategies. Performance variation between requests is due to these classes.
On a first approach, we instrumented the solver class for all metrics available in order to study the behaviour of each type of solving technique.
We tested these solvers by running three puzzles of each size with each of the available solving techniques (except for the 25x25 puzzle size, which only has 2 puzzles available) and an ample sample size of unsigned entries.
We did this to understand how the solver performance is affected by different puzzle sizes and number of unassigned entries. (this is not the only thing that affects performance but it is the only one we can control)
We save these results in JSON files and generate graphics from these results.
By analysing these graphics and calculating the correlation between Instruction Count (which approximately gives us the execution cost) and all other metrics, we came to the conclusion that most metrics show a correlation of >95%. With that in mind, we chose the metric with least amount of overhead to be the one instrumented during runtime, giving us a close approximation of the cost that puzzle will have on the system while giving us the least amount of overhead from instrumentation, giving a significant performance boost. With that said, we have picked the following metrics for each solving strategy:

BFS - Method Count
CP - New Count
DLX - New Array Count

We can see in annex A and B the values of the correlation between Instruction Count and the rest of the metrics for each strategy.

WebServer

The WebServer keeps an HashMap which associates each Thread solving a puzzle with the parameters of that puzzle.
In the instrumentation classes we also keep an HashMap that associates each Thread with the metrics for it's request.
When the thread finishes solving the puzzle, it saves the parameters of the puzzle with the associated metrics in a JSON file for future statistics by obtaining them from the HashMaps with their correspondent Thread ID.
Finally the entries of the current thread in the two HashMaps are removed. This way, the server does not mix metrics from different requests while being multi-threaded (since the server uses a cached thread pool which can reuse threads).

Future Implementation

Metrics Storage System

The collected metrics must be permanently stored so that the load balancer can access them and use them to decide where to forward each request. We plan to store the metrics obtained from each request in the dynamoDB.
To reduce the overhead, we plan to send the metrics in chunks, i.e send all metrics every X requests or only send when the server is not massively overloaded.
Alternatively, we could have instances dedicated to collecting metrics and sending them to dynamoDB, which would synchronize

among themselves to ensure that the dynamoDB stays always consistent.
Each register in the database will have the parameters of the puzzle, the values of the metrics obtained and the cost associated with that puzzle. When the Load Balancer receives a query with the same parameters, the LB can estimate more precisely the cost of the query.

Load Balancer

All correlations we have seen between metrics are linear. As such, we can assume the cost of each request to be the number of the instrumented metric for its respective strategy.
When the load balancer receives a request, if it is the first request of its kind (i.e a unique configuration of board and unassigned entries) the load balancer must assume the worst case, by choosing the cost of a request with similar parameters (we bootstrap this process by providing the experimental results already obtained).
The instrumented metric must be converted into raw instruction cost, which can be done by a simple multiplication for each algorithm.
The instruction cost for each strategy was obtained using the following formula:
$InstructionCost = \#ChosenMetric * \frac{\#Instructions}{\#Metric}$
Since the correlation is linear, the ration #Instructions/#Metric will be constant.
With that in mind, these were the obtained Instruction Costs for each strategy:

BFS:
$InstructionCost = \#Methods * 2.5 * 10^2$
CP:
$InstructionCost = \#NewCount * 3 * 10^2$
DLX:
$InstructionCost = \#NewCount * 2 * 10^6$

Otherwise, since the cost of each request is constant, we can know for sure what it will cost (the cost will be stored in dynamoDB).
The load balancer can then keep track of the load of each server and choose the least busy server for each request it receives. To decrease the accesses of the load balancer to the dynamoDB, we can cache responses in a LRU manner and use those to infer the cost of new requests.

Since requests can be long running, we must periodically update the cost of each server to account for requests that started a long time ago. This can be done by periodically subtracting the cost of each long running request of each server by a fixed amount (since it is expected to be a direct correlation between the instrumented metrics and the cost in CPU and time of each request).

Auto Scaling

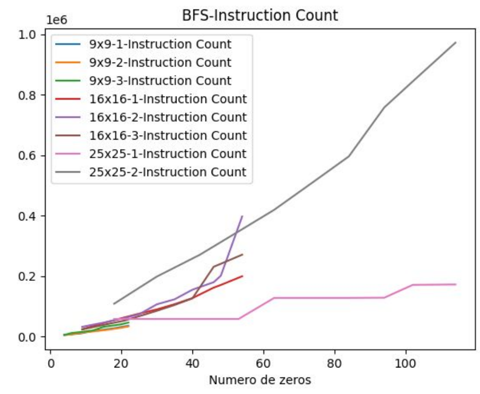
Since the load balancer can only distribute work between the running instances, it is the job of the auto scaler to detect when the computing nodes are overloaded and launch new instances to handle the demanding workload. Conversely, it must also shut down instances when demand is small to spare resources. Launching new instances is simple: as every instance in AWS has the same computational power, we just detect when the load on the system as a whole is higher than the capacity of the current running instances and launch new instances until we can cope with the current running cost. To scale down, we must take into account several factors to decide which instances to terminate:

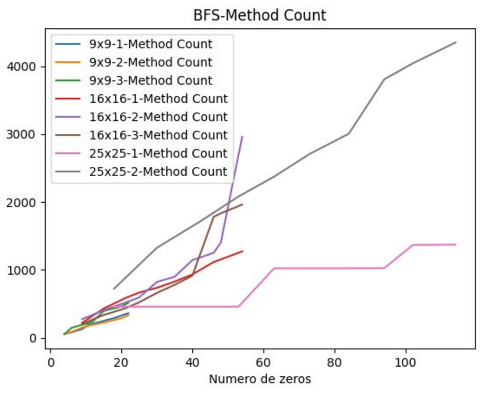
→ Longer running instances are more preferable than new instances, since they are more likely to have reduced performance (the program might open files it never closes or allocate memory that is never collected).

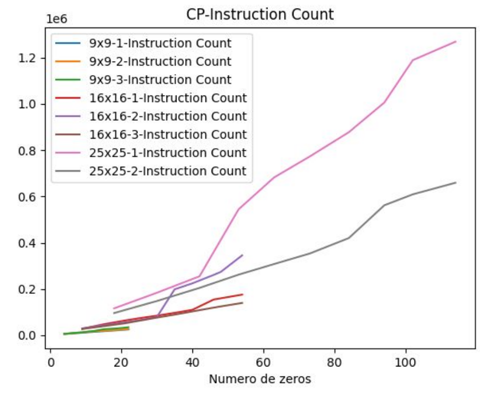
→ Instances with lower load are preferable, since we must for the instance to not be working before shutting it down or it's state will be lost.

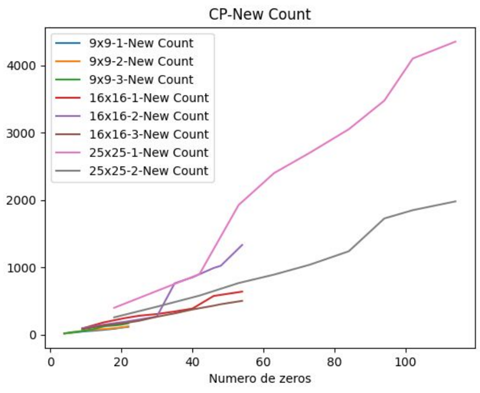
→ It is preferable to shutdown instances after spending the time that we paid for them, i.e it does not make sense to shutdown an instance after we just paid for it since we are charged the same amount. Conversely, if we plan on shutting down an instance we can give it work to do for it's current time slot, relaxing the load balancing for that instance but still ensuring that all requests will be replied before it's time slot expires and it is terminated.

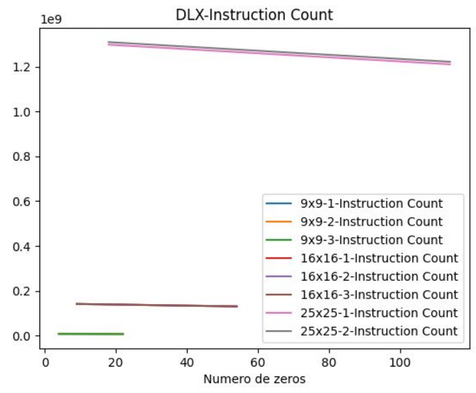
Annex A

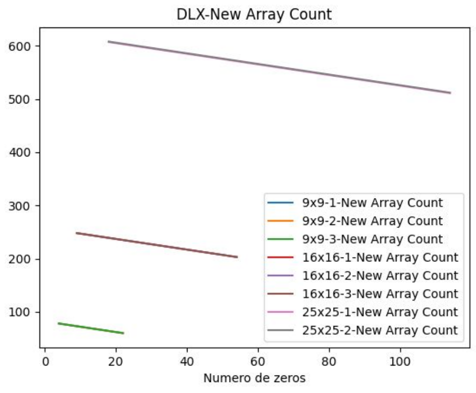

BFS-Instruction Count
1e6
1.0
0.8
0.6
0.4
0.2
0.0
9x9-1-Instruction Count
9x9-2-Instruction Count
9x9-3-Instruction Count
16x16-1-Instruction Count
16x16-2-Instruction Count
16x16-3-Instruction Count
25x25-1-Instruction Count
25x25-2-Instruction Count
0
20
40
60
80
100
Numero de zeros


BFS-Method Count
4000
3000
2000
1000
0
9x9-1-Method Count
9x9-2-Method Count
9x9-3-Method Count
16x16-1-Method Count
16x16-2-Method Count
16x16-3-Method Count
25x25-1-Method Count
25x25-2-Method Count
0
20
40
60
80
100
Numero de zeros


CP-Instruction Count
1e6
1.2
1.0
0.8
0.6
0.4
0.2
0.0
9x9-1-Instruction Count
9x9-2-Instruction Count
9x9-3-Instruction Count
16x16-1-Instruction Count
16x16-2-Instruction Count
16x16-3-Instruction Count
25x25-1-Instruction Count
25x25-2-Instruction Count
0
20
40
60
80
100
Numero de zeros


CP-New Count
4000
3000
2000
1000
0
9x9-1-New Count
9x9-2-New Count
9x9-3-New Count
16x16-1-New Count
16x16-2-New Count
16x16-3-New Count
25x25-1-New Count
25x25-2-New Count
0
20
40
60
80
100
Numero de zeros


DLX-Instruction Count
1e9
1.2
1.0
0.8
0.6
0.4
0.2
0.0
9x9-1-Instruction Count
9x9-2-Instruction Count
9x9-3-Instruction Count
16x16-1-Instruction Count
16x16-2-Instruction Count
16x16-3-Instruction Count
25x25-1-Instruction Count
25x25-2-Instruction Count
0
20
40
60
80
100
Numero de zeros


DLX-New Array Count
600
500
400
300
200
100
9x9-1-New Array Count
9x9-2-New Array Count
9x9-3-New Array Count
16x16-1-New Array Count
16x16-2-New Array Count
16x16-3-New Array Count
25x25-1-New Array Count
25x25-2-New Array Count
0
20
40
60
80
100
Numero de zeros

Annex B

	BFS	CP	DLX
Store Count	0.9765	0.9885	0.9999
Load Count	0.9999	0.9999	0.9999
Basic Block Count	0.9999	0.9999	0.9999
Branch Count	0.9999	0.9999	0.9999
Method Count	0.9800	0.9951	0.9602
Field Load Count	0.9998	0.9999	0.9981
Stack Depth	0.7319	0.6222	0.3056
New Count	0	0.9971	0.9638
Field Store Count	0	0	0.9160
New Array Count	0	0	0.9655