UPPSALA UNIVERSITY

CLOUD COMPUTING

Benchmarking of option price solvers as a service in OpenStack

---

Authors:

Andrea RYLANDER

Alessandro PICCOLO

Abdullah Al HINAI

October 21, 2016

UPPSALA
UNIVERSITET

***Abstract*—Mass migration from on-premise environment to Cloud Computing for legacy systems is coming inevitably in near future. In this paper, we present the integration of an existent project to a Cloud Computing environment by extending an application that was developed by the Computational Finance research group at the Division of Scientific Computing in Uppsala University. This integration is capable of launching the application as a public service on Cloud, by using the contextualization feature with Docker containers to minimise the set-up complexity and support Interoperability, which is the ability to exchange and make use of information between different cloud types and vendors to avoid vendor lock-in.**

I. Introduction

THE purpose of this project was to create a cloud based system as a service such that the user could through a web-interface start different benchmarks of option price solvers. The idea derives from the project, BENCHOP the BENCHmarking projects in option pricing [1]. Basically it should act as a platform for testing which option is the fastest (execution time) and most accurate (relative error) such that different solvers can be evaluated with the same premise.

To truly understand the problem one would have to know what is an option. In finance, an option is a contract which gives the buyer the right, but not the obligation, to buy or sell an underlying asset or instrument at a specified strike price on or before a specified date, depending on the form of the option. To not create an arbitrage (earn risk free money) one has to set an appropriate price to the option. By solving the Black-Scholes equation (1), one would get the arbitrage free option price. However this is sometimes, depending on the option type, not possible to do analytically hence you have to use numerical methods. Equation 2 shows the stock price changes, variable S is the stock price, r is the interest rate, σ stands for the volatility (changes in underlying asset S), dt is the time stepping and lastly dW is the Weiner-process (random normal distributed number). The BS equation 1 is an partial differential equation (PDE) variable u is the option price and the rest is as before. For a European call, American call and up-and-out options on an underlying stock paying no dividends, the equations are:

$$\frac{\partial u}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 u}{\partial S^2} + rS\frac{\partial u}{\partial S} - ru = 0 \tag{1}$$

$$dS = rSdt + \sigma SdW \tag{2}$$

Basically we have different option types which can be solved with different numerical methods and it is also important to know that one can change the parameters such as the volatility, strike price and final time. In our system we preformed our benchmarks on problem 1a (European), 1b (American) and 1c (Up-and-Out) they are different option types with a specific set of parameters. By changing variable ”problems” i the main.py file one can choose which problems to solve. There are detailed description of the problems and more problems described here [2]. Each problem was solved with two numerical methods, the COS method and finite difference method for our tests. The system is easily scalable by adding more folders containing option price solvers to the github repository [3], it will automatically use the newly added folders. Different types of numerical methods can be found here [2]. When initializing a benchmark test one can set the σ parameter or else it will use the default value of 0.15. To exploit the cloud capabilities the different problem types were executed as tasks and split up in between the pool of workers. The results is collected by the flask server and visualized with the help of a python library called Pygal.

The most interesting part of the system is to run the different solvers on the same hardware to get a comparable execution time. The system does not use MATLAB since it is an licensed software, instead it uses GNU Octave (open source implementation of the MATLAB) which is much slower.

II. Overview of the model

Our model is comprised mainly of four components: a RabbitMQ server [9], the flask framework [6] and two celery workers [8]. This section will explain how these different components work and co-operate in our application.

A. Server

In the current state of the project the server is a RabbitMQ server running on a virtual machine with an Ubuntu 16.04 operating system. The server makes use of the flask framework to create a web interface for the user. The user can connect to our application through this web interface in the following manner: http://<IP of server>:5000/benchmark. It is also possible for the user to start the application with a parameter, at the moment this parameter is σ, so the user can specify with which values for σ she/he wants to run the application. To do this the user simply adds the following to the above URL: /<desired value for σ>. When the user connect to the web interface a request to start the application is sent to the server. The number of tasks started depends on the number of problems specified in the code. The server has a distributed setup, which means we specified a user and a vhost to allow multiple workers to connect to the same RabbitMQ broker. The tasks are then distributed as evenly as possible to the workers connected. As stated the RabbitMQ server uses a vhost to allow several workers to connect to it, creating a worker pool. It is to this worker pool received tasks are distributed.
When all the tasks are sent, the server awaits the results from each task and saves the results. For this purpose we also needed to specify a backend. Once all tasks are completed and the server has all the expected results it will process the results in order to be able to present it to the user in a clear manner. We chose to present the results to the user as bar charts using the library pygal [7] The last thing the server does it to render a HTML-page, where these bar charts are sent as arguments in order for them to be rendered in the HTML-page and the user can now see the results.

B. Workers

We wanted our model to be as lightweight as possible, and one of our goals was to produce a model that was

infrastructure independent, which is why part of our model is based on Docker. Although the base infrastructure of our server and two workers is the same, i.e a virtual machine running an Ubuntu 16.04 operating system, the two workers are actually Docker containers. Meaning, given that a machine has the docker-engine installed on it, it can easily pull the docker image for the worker from docker-hub and then simply run this image as a container and a worker is ready to be used.

C. Docker setup

To be able to set up the workers to run as docker containers we first had to create a docker image. This image is created by a file explaining everything this images is to contain, this file is called *Dockerfile*.

The *Dockerfile* starts by specifying which base image to use. We chose to use Ubuntu as our base image. Then it proceeds with specifying what is to be installed and commands to be run. One also specifies a work directory, this means that when the image is created and later run as a container, it is from this directory one starts to work in. Lastly one can specify what commands, can also be programs/scripts, to run when the image is run. In other words, what the images is supposed to do as soon as it has been ”booted”.

Once the *Dockerfile* is done one can start with creating the image by typing the following command, $sudo docker build -t <name of the image> /path/to/Dockerfile. If one is standing in the folder where the *Dockerfles* is stored, then one simply uses a dot, ”.”. This dot tells docker that the files resides in the current folder. Once created, the images needs to be pushed to docker-hub. To do this one is required to have a docker-hub account since pushing the image is not possible until after one has logged in to such account. When pushing the image one specifies the username and the name of the repository the image is to be pushed to. If ones user name is *docker-user* and one wishes to push the image to the repository *docker-image* then the command would be, $sudo docker push docker-user/docker-image. When someone then wants to use this image, that person simply needs to pull the images from docker-hub with the pull command.

It is important to note that this person needs to be running a machine with docker installed to be able to pull the image. To run an image one uses dockers run command followed by the name of the image to be run. If one would want to use the image we use for or workers one can pull the image ”andreeeea/group11-benchop” and run it. One should also specify from which port one wishes to access the container with the flag ’-p’. More information on how to use docker can be found here [4]

Figure 1 represents how the different components in our application work together. Figure 2 and 3 show the results as bar charts while figure 4 shows them as plain text. We figured it could be useful to have both representations of the results.

D. Contextualization

We wanted to automate as much as possible in this project, so we have contextualization files for all the different components of our model. In our github [3] repository we have

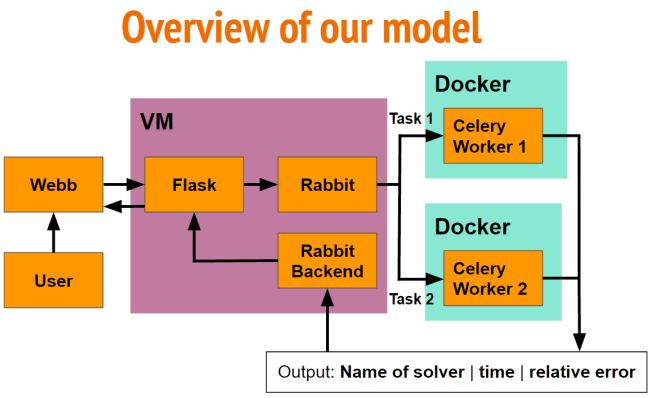

Fig. 1. Overview of different OpenStack components in backend system

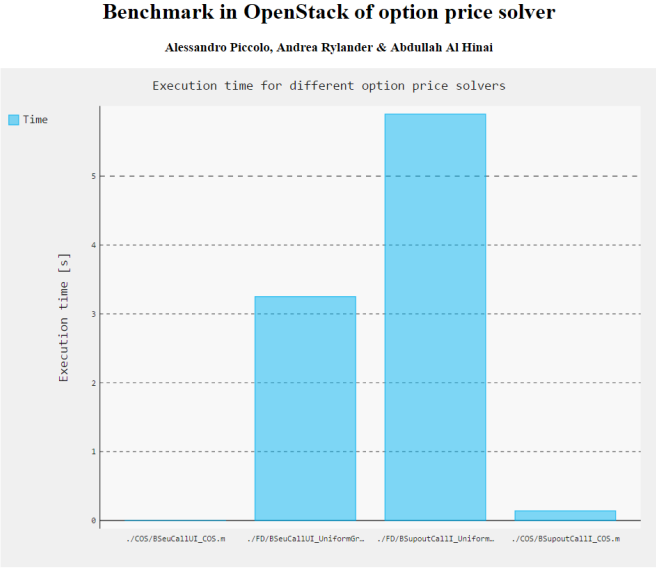

Fig. 2. Visualization of output time with flask and pygal

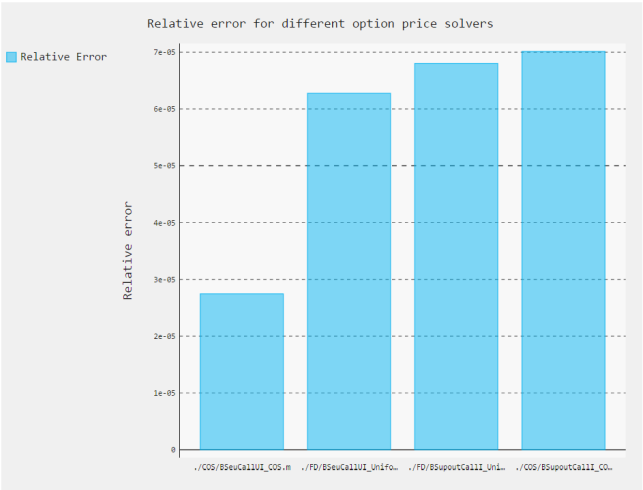

Fig. 3. Visualization of output relative error with flask and pygal

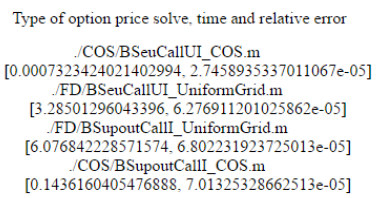

Fig. 4. Lastly in webb page shows extra text output of benchmarks

two contextualization folders, one is called *contextualization-server* and the other *contextualization-docker*. They contain all the files that one needs to create the server and the workers respectively. Each folder contains a python script that connects to the proper OpenStack APIs and creates an instance with the specified flavor and with the help of the cloud-init file, which also resides in the folder, installs all the required programs. For example the cloud-init file for the server includes the installation of flask and pygal among other things, while the cloud-init file for the workers installs docker. The workers also needs for example celery, since they are celery workers, but this is not specified in the cloud-init file. This is because once one spawns a machine with the files in the contextualization-docker folder, the cloud-init file is specified to pull the docker image we created for our application. This image contains all the programs and libraries needed to run the workers.

III. Issues with the model

As mentioned earlier we wanted our solution to be as automated and lightweight as possible, which is why we chose to run the workers as docker containers, but we actually intended to run the server as a docker container as well. The reason our application does not do that at moment is because it requires the correct port-setup on all the machines for the workers to be able to connect to the RabbitMQ server. Given more time we would have been able to figure out how to configure the ports correctly so that server and workers could connect properly.

The user can start the application with a parameter representing a value for sigma with which the user wishes to run the application. This actually requires the user to know what values are acceptable to pass as a sigma value. We treated the MATLAB code given to us as a black-box, we did not change the code in anyway. Meaning that we do not verify the passed parameter, we just pass it directly to the black-box. What one could do is have the server verify if the parameter passed by the user has a value within an acceptable range, and if the parameter is not accepted ask the user to enter a value within in the accepted range or just ignore the passed parameter and run the application with a default value. One would of course let the user know that the application was run with another value than the one given. This solution is a bit similar to the way we handle the case where the user chooses not to pass a parameter. In that case we use a default value for sigma. Another solution would be to pass the parameter from the user directly to the MATLAB code and modify the code to verify the value before running. Anyway the concept of passing an input parameter is implemented.

When it comes to the web interface we do not use the most elegant solution. Once the user starts the application, i.e send a request to the server, all the user sees is a blank page. The user has to wait for server to receive the results from all the tasks and then create the bar charts before finally rendering a HTML-page with the results. It would be more elegant to directly render a HTML-page which would tell the user that the server is processing the request, tell the user if something went wrong, how longtime she/he needs to wait for the results or at least show a loading screen. This was not a priority due to the lack of time, but it would be easy to implement with for example javascript in the HTML-page.

A. Future endeavors

The Cloud Computing benchmarking API has been conducted in OpenStack educational environment using low range of resources, flavors and few Celery workers working in parallel, limited by available instances. It will be interesting to deploy the system on high-end virtual machines with at least six standalone running workers corresponding to the number of problems to be solved. It would be interesting from a computational finance point of view to see how fast different solvers are when varying the hardware. Example some solutions are heavily dependent on RAM memory, such as finite difference methods.

In addition, due to the time limitation of this project, three weeks, we implemented one among multiple ideas to run this service as the optimal solution for option pricing. Given more time, we would have wanted to expand our application to be able to multiple Celery workers in parallel for each available method within each problem to be solved and computed. In other words, each problem would be handled by one worker, and that worker would start other workers to compute the results of each numerical method separately in parallel. This will make the service faster and reduce the computational workflow, while also reducing the time of consuming cloud resources. Thus leading to reduction of expenses for the Cloud vendor in the scenario of the system being a real product. In addition to that, the solution can be extended to provide the user the ability to enter more parameters to the problems to be computed via REST API/interface, and give the user the full functionality of uploading any new methods to be involved in the problem evaluation and benchmarking.

IV. Conclusion

In conclusion, we presented development of benchmarking software as a service based on Cloud Computing environment, out of existent in-house legacy system. This development expedites the evaluation of the benchmark by running different Celery workers in parallel in the backend. The system has the capability to benchmark common problems (equations) with a set of heterogeneous methods to give a set of option pricing that are used for evaluation and comparisons purposes. The application uses a wide range of OpenSource tools to minimize expenses of movement to Cloud environment, and

to be interoperable with any other system. We also presented the designing layers of our development model and how the system isolated from the operating system resources by adding Docker engine containers,thus supporting lightweight OS-level virtualization and enables deploy ready-to-run portable software. In addition , we demonstrated the capabilities of Cloud Computing environment to host existent on-premise computational APIs.

REFERENCES

[1] von Sydow, L., Josef Hk, L., Larsson, E., Lindstrm, E., Milovanovi, S., Persson, J., Shcherbakov, V., Shpolyanskiy, Y., Sirn, S., Toivanen, J., Waldn, J., Wiktorsson, M., Levesley, J., Li, J., Oosterlee, C.W., Ruijter, M.J., Toropov, A. and Zhao, Y. (2015) BENCHOP the BENCHmarking project in option pricing, International Journal of Computer Mathematics, 92(12), pp. 23612379. doi: 10.1080/00207160.2015.1072172.

[2] The original BENCHOP - The BENCHmarking project in Option Pricing http://www.it.uu.se/research/project/compfin/benchop/original

[3] Project github https://github.com/AndreaRylander/Cloudgroup11

[4] https://docs.docker.com/engine/tutorials/dockerimages/

[5] Docker image https://hub.docker.com/r/andreeeea/group11-benchop/

[6] Flask documentation http://www.flaskapi.org/

[7] Pygal documentation http://pygal.org/en/stable/

[8] Celery documentation http://docs.celeryproject.org/en/latest/reference/

[9] Rabbit documentation http://www.rabbitmq.com/documentation.html